https://doi.org/10.35662/unine-thesis-3199

Université de Neuchâtel
Institut d'informatique

WebAssembly as an Abstraction for Secure and Efficient Computing in Trusted Execution Environments

Ph.D. dissertation

submitted to the Faculty of Science of the University of Neuchâtel

to attain the degree of Doctor of Philosophy in Computer Science

by

Jämes Ménétrey

Approved by the dissertation committee:

Prof. Dr. Pascal Felber • thesis co-director • University of Neuchâtel, Switzerland
Prof. Dr. Marcelo Pasin • thesis co-director and reviewer • HES-SO, Switzerland
Prof. Dr. Rüdiger Kapitza • reviewer • Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
Prof. Dr. Rachid Guerraoui • reviewer • École Polytechnique Fédérale de Lausanne, Switzerland
Prof. tit. Dr. Valerio Schiavoni • examiner • University of Neuchâtel, Switzerland

Thesis defended on November 4, 2024
Neuchâtel, Switzerland

unine
Université de Neuchâtel
Faculté des sciences

Rue Emile-Argand 11
CH-2000 Neuchâtel
doctorat.sciences@unine.ch

IMPRIMATUR POUR THESE DE DOCTORAT

La Faculté des sciences de l'Université de Neuchâtel autorise l'impression de la présente thèse soutenue par

Monsieur Jämes MENETREY

Titre :

"WebAssembly as an Abstraction for Secure and Efficient Computing in Trusted Execution Environments"

sur le rapport des membres du jury composé comme suit :

- **Prof. Pascal Felber,** directeur de thèse, Université de Neuchâtel, Suisse
- **Prof. Marcelo Pasin,** Haute école spécialisée de Suisse occidentale, Suisse
- **Prof. Rüdiger Kapitza,** FAU Erlangen-Nürnberg, Allemagne
- **Prof. Rachid Guerraoui,** École polytechnique fédérale de Lausanne, Suisse
- **Prof. tit. Valerio Schiavoni,** Université de Neuchâtel, Suisse

Neuchâtel, le 13 novembre 2024

Le Doyen, Prof. P. Brunner

To my parents, Brigitte and Bernard.
To my brother, Steven.
To my darling, Laetitia.
To my loved animals, Kéké, Choki, Wicky, Keyla, Oréa, Gybee and Minimoy.
Thank you for all of your support and for always trusting me along the way.

Abstract

The rapid evolution of computing infrastructure has led to the emergence of the *cloud-edge continuum*, a seamless integration of cloud, edge, and IoT devices. This paradigm shift enables applications to leverage the strengths of each computing environment, from the vast resources of the cloud to the low-latency processing of edge devices and the ubiquitous presence of IoT sensors. However, the heterogeneity of the cloud-edge continuum poses significant challenges in terms of software development, deployment, and security.

The shift towards cloud and edge computing has introduced new security challenges for customers and providers. Customers must trust cloud providers to adequately isolate workloads and protect sensitive information, while providers must securely execute untrusted customer software. Confidential computing has emerged as a promising approach to address these security challenges, leveraging trusted execution environments (TEEs) to protect sensitive data and code from unauthorised access, tampering, or disclosure. However, the current confidential computing ecosystem remains fragmented, with CPU vendors offering fundamentally different proprietary TEE implementations, which limits portable and secure application development.

To tackle the challenges of the heterogeneous cloud-edge continuum and the shortcomings of current TEE solutions, this thesis advocates adopting the open standard WebAssembly (Wasm) as a lightweight, efficient, and secure portable compilation target. Wasm's platform-agnostic design and sandbox make it an ideal candidate for bridging different computing environments while providing a robust foundation for secure execution. The combination of Wasm and TEEs establishes a mutually beneficial security model as they protect sensitive data and computations from unauthorised access while shielding the underlying infrastructure from malicious code.

This thesis presents a comprehensive approach for secure and efficient application execution in TEEs. The first research work reviews attestation principles and compares how modern TEEs leverage these mechanisms, revealing that existing solutions widely differ in maturity and security. These insights form the basis for the design and implementation of trusted Wasm runtimes, attestation mechanisms, and communication systems developed throughout the dissertation.

The second research of this thesis demonstrates the practical benefits of using Wasm for client-side cryptography in browsers and shows how it outperforms conventional JavaScript implementations for IncaMail, the Swiss Post's secure email service. Furthermore, Wasm is also used to reduce the trusted computing base of IncaMail, enabling an honest-but-curious model.

Based on the promising results of running Wasm in browsers, the third research work introduces Twine, a trusted runtime for running Wasm-compiled applications within Intel SGX, a TEE implementation for creating secure environments named *enclaves*, establishing a two-way sandbox. Twine leverages memory safety guarantees of Wasm and abstracts the complexity of TEEs, supporting the execution of legacy and language-agnostic applications.

Building upon Twine, the fourth research work introduces WaTZ, an efficient and secure runtime for trusted execution of Wasm applications on Arm TrustZone, a TEE commonly found in

small and edge devices. WATZ includes a lightweight attestation system optimised for Wasm applications running in TrustZone, addressing the lack of built-in attestation mechanisms.

The fifth and final work of this dissertation presents a portable and mutually attestable publish/-subscribe system for trustworthy communication in the cloud-edge continuum. Leveraging the abstraction provided by trusted Wasm runtimes, the same application can run on different CPU architectures and be attested on various TEE platforms through standard attestation primitives. As a result, the proposed solution establishes secure communication channels between mutually distrusting remote parties without being bound to a particular CPU vendor.

Confidential computing has emerged as an increasingly important trend for IoT, edge, and cloud environments. This dissertation leverages Wasm to address the challenges of programming, deploying, and establishing trust in these complex trusted environments. Wasm's lightweight specifications, large support of programming languages, near-native execution speed, robust sandbox, and versatile system abstraction make it an ideal candidate for envisioning the cloud-edge continuum as an interoperable, scalable, and distributed system. It becomes clear that Wasm and trusted computing can be the bedrock for software development in large-scale systems in the coming years, transforming the development lifecycle of future applications.

Keywords: trusted execution environments (TEEs), confidential computing, attestation, cryptography and security, WebAssembly, Intel SGX, Arm TrustZone.

Résumé

L'évolution rapide des infrastructures informatiques a conduit à l'émergence du *continuum cloud-edge*, qui représente l'intégration continue et transparente d'appareils déployés dans trois environnements initialement séparés : le *cloud* (centres de données centralisés distants), la périphérie ou *edge* (infrastructures proches des utilisateurs finaux), et l'Internet des objets ou *IoT* (appareils connectés distribués localement). Ce changement de perspective permet aux logiciels de tirer parti des avantages de chaque environnement informatique, qu'il s'agisse des vastes ressources du cloud, du traitement à faible latence en périphérie ou de l'omniprésence des capteurs de l'Internet des objets. Cependant, le caractère hétérogène du continuum cloud-edge représente un défi de taille au niveau du développement logiciel, du déploiement et de la sécurité.

L'avènement de l'informatique dans le cloud et en périphérie a entraîné l'émergence de nouveaux défis de sécurité pour les fournisseurs de ces infrastructures et leurs clients. Cette relation implique une confiance mutuelle : les utilisateurs comptent sur les fournisseurs pour isoler adéquatement leurs charges de travail informatiques et protéger leurs informations sensibles, tandis que ces derniers doivent exécuter des logiciels clients de manière sécurisée, sans garantie de leur fiabilité. L'informatique confidentielle a émergé comme une approche prometteuse pour répondre à ces défis sécuritaires, tirant parti des environnements d'exécution de confiance (TEE) pour mieux protéger les données et les applications sensibles contre l'accès non autorisé, la falsification ou la divulgation de ces dernières. Toutefois, l'écosystème actuel de l'informatique confidentielle demeure fragmenté, les fabricants de processeurs proposant des implémentations propriétaires et fondamentalement différentes de ces environnements d'exécution de confiance, ce qui limite la portabilité et la sécurité des applications.

Pour relever les défis du continuum cloud-edge qui est par essence hétérogène, ainsi que de combler les lacunes des solutions actuelles que constituent les environnements d'exécution de confiance, cette thèse de doctorat préconise l'adoption du standard ouvert *WebAssembly* comme format de compilation portable, léger, performant et sécurisé. WebAssembly a été conçu pour réduire son couplage aux plateformes d'exécution et pour offrir un environnement d'exécution isolé, lui conférant des atouts indéniables pour devenir une référence en matière d'exécution sécurisée. La combinaison de WebAssembly et des environnements d'exécution de confiance offre une sécurité mutuellement bénéfique, car elle protège les données sensibles et les calculs informatiques des clients contre les accès non autorisés, tout en protégeant les fournisseurs d'infrastructure contre les programmes malveillants de ces mêmes utilisateurs.

Cette thèse présente une approche globale pour l'exécution sécurisée et performante d'applications dans les environnements d'exécution de confiance. Le premier travail de recherche examine les principes d'attestation et compare la façon dont ces environnements modernes exploitent ces mécanismes, révélant que les solutions existantes diffèrent considérablement en termes de maturité et de sécurité. Ces observations constituent le fondement de la conception et de l'implémentation de TEE basés sur WebAssembly, de mécanismes d'attestation et de systèmes de communication développés tout au long de cette thèse.

Le deuxième travail de recherche de cette thèse démontre les avantages pratiques de l'utilisation de WebAssembly pour la cryptographie, dont son exécution est déléguée aux navigateurs Web et montre comment cette technologie surpasse les implémentations JavaScript conventionnelles pour IncaMail, le service de messagerie sécurisé de La Poste suisse. WebAssembly est également utilisé pour réduire la base de confiance informatique (TCB) d'IncaMail, ce qui permet un modèle de confiance honnête mais curieux (honest-but-curious).

En se basant sur les résultats prometteurs de l'exécution de WebAssembly dans les navigateurs Web, le troisième travail de recherche présente TWINE, un environnement d'exécution de confiance permettant l'exécution d'applications compilées en WebAssembly au sein d'Intel SGX, une implémentation de TEE créant des environnements sécurisés appelés *enclaves*, établissant ainsi un environnement d'exécution isolé bidirectionnel. TWINE exploite les garanties de sécurité mémoire de WebAssembly et fait abstraction de la complexité de Intel SGX, prenant en charge l'exécution d'applications existantes et indépendantes du langage de programmation.

En s'appuyant sur TWINE, le quatrième travail de recherche détaille WATZ, un environnement d'exécution de confiance pour les applications WebAssembly dans Arm TrustZone, un TEE couramment présent dans les petits appareils et appareils de périphérie. WATZ inclut un système d'attestation léger et optimisé pour les applications WebAssembly, répondant au manque de mécanismes d'attestation intégrés à cet environnement d'exécution de confiance.

Le cinquième et dernier travail de cette dissertation développe un système de publication/abonnement (publish/subscribe) permettant l'échange de données dont les acteurs sont mutuellement attestés dans le continuum cloud-edge. En exploitant l'abstraction fournie par les environnements d'exécution de confiance basés sur WebAssembly, une même application peut s'exécuter sur différentes architectures de processeurs et être attestée sur plusieurs implémentations de TEE grâce à des primitives d'attestation standardisées. En conséquence, la solution proposée établit des canaux de communication sécurisés entre des parties distantes qui se méfient mutuellement, sans être tributaire d'un fabricant de processeurs spécifique.

L'informatique confidentielle s'est imposée comme une tendance de plus en plus importante pour l'Internet des objets et les machines à la fois situées en périphérie ou dans le cloud. Cette thèse de doctorat exploite WebAssembly pour répondre aux défis de programmation, de déploiement et d'établissement de confiance dans ces environnements de confiance complexes. WebAssembly se distingue par ses spécifications simples, sa capacité à être intégré à un large éventail de langages de programmation, sa vitesse d'exécution comparable à celle des applications natives, son environnement isolé robuste et son abstraction de la plateforme d'exécution. Ces caractéristiques en font un choix idéal pour concevoir un système distribué, évolutif et interopérable, allant du cloud à l'Internet des objets. Il devient évident que WebAssembly et l'informatique confidentielle peuvent constituer un nouveau socle solide pour le développement logiciel dans les systèmes de petite à grande échelle au cours des années à venir, transformant ainsi la façon dont les applications futures seront conçues et développées.

Mots-clés : environnements d'exécution de confiance (TEE), informatique confidentielle, attestation, cryptographie et sécurité, WebAssembly, Intel SGX, Arm TrustZone.

Acknowlegements

Completing this doctoral thesis is undoubtedly one of the most demanding challenges I have ever accomplished, yet it remains profoundly rewarding, not merely through the acquisition of technical expertise and research experience, but through lessons in humility, collaboration, and the fundamental importance of human connection in general. While genuine passion for computer science provided the initial momentum, I strongly believe that no one can make it from naive enthusiasm to finished dissertation without a tremendous amount of support, both intellectual and emotional. The privilege of being surrounded by exceptional individuals, from supervisors and colleagues to friends and family, has made this journey possible.

My deepest gratitude extends to my supervisors, Pascal Felber and Marcelo Pasin, whose unwavering support, expert guidance, and remarkable positivity have been indispensable throughout this research. Their intellectual insights and visionary perspectives, combined with the academic freedom they afforded, have been instrumental in shaping the start of my scholarly career. I shall always treasure memories of our journeys to conferences, from neighbouring to distant countries, where scholarly presentations evolved into profound shared experiences. Special appreciation is also due to my colleague Valerio Schiavoni, who provided invaluable support for my technical contributions while instilling a strong sense of rigour and discipline in my approach. The Italian culinary experiences he graciously shared during our conferences added genuine warmth to our scholarly exchanges. As the adage suggests, while the journey matters more than the destination, the companions who accompany us prove equally significant. Though this particular chapter concludes, future fruitful collaboration undoubtedly awaits!

I owe much appreciation to the external members of my thesis jury, namely Rüdiger Kapitza and Rachid Guerraoui, for the time and energy they dedicated to evaluating this doctoral dissertation, as well as for the insightful discussions during and after the viva.

I am also grateful to the international research partners and companies that got involved in my dissertation. The Horizon European project VEDLIoT provided funding and an extraordinary fertile soil for innovation. In the context of this project, I am grateful to Jens Hagemeyer and Carola Haumann for their outstanding leadership of the project, as well as to Alysson Bessani, Antonio Casimiro, Pedro Trancoso, Nils Kucza and the other VEDLIoT members for their hard working contributions and pleasant moments during the general assemblies. I am also particularly thankful to Credora and its members, Giovanni Mazzeo, Arne Hollum and Darshan Vaydia, for funding and contributing to the work presented in this thesis. I highly appreciated our discussions, collaboration, and the remarkable results that emerged from this joint effort.

I extend a particular recognition to Raphaël Barazzutti, without whom this doctoral journey might never have commenced. As my undergraduate instructor, he generously facilitated the connection with Pascal, providing me with an extraordinary opportunity to pursue doctoral research. Raphaël, thank you for your kindness, all the lectures where every student could feel that you are truly passionate about them, and our long night discussion outside the HEIG-VD!

This thesis would not have been possible without the inexhaustible support of PhD students, to whom I owe immense gratitude. Upon my arrival at the institute, Sébastien Vaucher, Rémi Dulong, Peterson Yuhala, Christian Göttel, Isabelly Rocha, Dorian Burihabwa, Catherine Ikae and Rafael Pires welcomed me with open arms. They made sure I had everything needed to begin this journey, and their kindness transformed what could have been an overwhelming start into an exciting adventure. Sébastien and Rémi deserve special thanks for the times we shared managing the IIUN cluster together. Those days spent configuring servers and troubleshooting systems remain some of my favourite memories. Given the chance, I would gladly relive those sessions! Peterson became far more than a colleague through our countless discussions and collaborative work. Our joint exploration of Intel SGX retrofitting led to both breakthrough moments and memorable frustrations. Those debugging sessions hunting down *segfaults* caused by troublesome `OCALL`s taught me as much about patience as they did about secure computing! Christian brought invaluable insights to our many conversations and good memories at the university and winter schools, which both led to published articles! As I prepare to leave, I take great comfort knowing the next generation of PhD students carries on the work. Romain De Laage and Abele Mãlan have taken over the IIUN cluster management, which could not be in better hands. Pasquale De Rosa, Louis Vialar, Simon Queyrut, Victor Villin, Mpoki Mwaisela, Andreas Athanasopoulos, Jakub Tluczek, Hortence Yiepnou, Elif Yilmaz and Vladimir Macko for the great memories and so much fun working with you on papers and assisting lectures.

My doctoral journey was made richer by brilliant colleagues who became sources of inspiration. Hervé Sanglard, for his kindness, entrusting his security lectures to me when I was starting out and for his positive attitude. It was really appreciated! Baptiste Lepers for instiling in me great ways to position and write papers, as well as for many thoughtful discussions, and engaging in pogo's during metal concerts; looking forward to our next one! Diana Ghinea brought endless laughter to our time in Tokyo and winter schools, along with her uncanny ability to predict the future with cards. Lorenzo Leonini for sharing many of your passions. One can only hope to have a fraction of your patience! Peter Kropf always knew precisely when to offer wise guidance, delivered with such care and consideration. Lydia Chen created wonderful memories both in and out of the lab, including those unforgettable homemade sushi rolls!

I have been fortunate to enjoy the friendship of many people who always believed in me. First and foremost, Jonathann, Anthony and Cintia Velen, with their continuous support and presence in my life. The countless dinners, evenings filled with laughter, concerts that lifted our spirits, and games that brought pure joy (or rage) created a sanctuary away from the pressures of research. Your friendship means the world to me! Romain Claret for your support since our master's studies and later for joining me in the doctoral adventure in Neuchâtel. Michael Barbey and Mathias Blank brought levity to our nerdy discussions and shared a sense of humour. Jonas Tachet, Lionel Perolini, Martial Gaillard, Jonathann and Anthony transformed our *quality control* visits to Irish pubs into traditions. Those evenings spent making sure Guinness standards remained acceptable across various establishments created memories that still bring smiles during challenging times! Céline Papin, Sébastien Pasteur, Maude Fischer and Axel Künzler turned ordinary weekends into adventures from spontaneous Verbier excursions to that memorable trip to Nézignan-l'Évêque. Lancelot Zeberli for the quality time we shared together. Timothée Léchot, Pierre-Emmanuel DuPasquier and Nathalie Vuillemin for our collaboration on Les Herbiers de Jean-Jacques Rousseau. This SNSF project demonstrated how enjoyable

interdisciplinary work can be with the right team. Anne-Claude Mühlberg for her care, boundless positivity and wise counsel. Finally, Grégoire Bovy, Frédéric Medana and Pascal Bangerter guided my first steps into the professional world of computer science during my apprenticeship.

None of this could have happened without my family. My parents have been my greatest champions from the very beginning. Throughout my life, they encouraged me to chase my interests and dreams while providing the safety net that allowed me to take risks. Their faith in me, sometimes blind, always absolute, has been the bedrock upon which everything else was built. This unconditional belief became the fuel that powered me through every challenge and setback. For this gift of endless support and love, I am forever grateful. Love you more than words can express, Mum and Dad. My brother Steven holds a unique place in this journey. Despite being younger, he has always looked after me with remarkable care and wisdom beyond his years. Steven, you are precious not just to me but to everyone fortunate enough to know you. Having you as my brother remains one of my greatest blessings. I would like to extend my gratitude to the family members who also supported me during my thesis: Arlette and Dominique Liardon, my in-laws with Vittoria, Bernard, Raphaël, Corine, Liliane and Roger Magnenat. You all welcomed me with open hearts and treated this thesis as your own achievement to celebrate.

Finally, my deepest gratitude belongs to Laetitia Magnenat, my partner and my anchor throughout this journey and beyond. Her boundless patience and loving support sustained me through every challenge this journey presented. Words cannot adequately express my appreciation, though I am grateful to have a lifetime ahead to attempt to do so. You are my everything!

Contents

Abstract vii

Résumé ix

Acknowlegements xi

List of acronyms xix

List of figures xxi

List of tables xxii

1 Introduction 1
- 1.1 The landscape of cloud, edge and confidential computing . . . 2
- 1.2 A brief history of portable compilation targets . . . 4
- 1.3 Research challenges . . . 5
  - 1.3.1 Abstracting the complexity of multi-paradigms TEEs . . . 5
  - 1.3.2 Attestation and communication between mistrusted parties . . . 7
- 1.4 Thesis contributions . . . 9
- 1.5 Thesis outline . . . 10
- 1.6 Attribution of contributions . . . 11
- 1.7 Published papers . . . 12
- 1.8 Open-source software contributions . . . 14

I Background 17

2 Trusted execution environments 19
- 2.1 A hardware-assisted security system with many faces . . . 20
- 2.2 A structured comparison of TEE characteristics and features . . . 22
  - 2.2.1 Introduction to Intel SGX . . . 23
  - 2.2.2 Introduction to Arm TrustZone . . . 27
  - 2.2.3 Introduction to AMD SEV . . . 30
  - 2.2.4 Introduction to other notable TEE implementations . . . 34
  - 2.2.5 A summary of the TEE characteristics . . . 36
- 2.3 Are TEEs the silver bullet for security? . . . 36

3 Attestation in confidential computing 39
- 3.1 Bringing trustworthy execution with TEEs . . . 40
- 3.2 Architecture and terminology . . . 40
- 3.3 Attestation types . . . 44

3.4 Overview of attestation for Intel SGX . . . 45
3.4.1 Local attestation . . . 45
3.4.2 Remote attestation: Enhanced Privacy ID (EPID) . . . 46
3.4.3 Remote attestation: Data Center Attestation Primitives (DCAP) . . . 48
3.5 Overview of attestation for AMD SEV . . . 50
3.5.1 Attestation with AMD SEV and SEV-ES . . . 50
3.5.2 Attestation with AMD SEV-SNP . . . 51
3.6 Portraying the need for an agnostic attestation abstraction . . . 52

4 WebAssembly 55
4.1 An agnostic standard for the cloud-edge continuum . . . 56
4.2 WebAssembly under the hood . . . 57
4.2.1 Bringing abstraction to programming languages . . . 58
4.2.2 Bringing abstraction to system environment . . . 58
4.2.3 Bringing abstraction to execution modes . . . 60
4.2.4 Bringing abstraction to operating system services . . . 61
4.3 A building block for interoperable TEE architectures . . . 63
4.4 A rich ecosystem . . . 65
4.4.1 An overview of the standalone runtimes landscape . . . 65
4.4.2 A summary of runtimes characteristics . . . 67
4.4.3 A balance of performance and lightweightness for TEEs . . . 67

II WebAssembly as an efficient and attestable abstraction 69

5 A use-case of WebAssembly: IncaMail 71
5.1 Introduction . . . 72
5.2 Background . . . 72
5.3 Use-case: IncaMail . . . 73
5.3.1 Offloading cryptographic operations . . . 74
5.3.2 Proposed workflow . . . 75
5.4 Evaluation . . . 77
5.4.1 Experimental setup and methodology . . . 77
5.4.2 Micro-benchmarks: encryption scheme . . . 77
5.4.3 Scalability of the encryption cipher . . . 78
5.5 Synthesis and next steps . . . 79

6 A trusted WebAssembly runtime for Intel SGX: Twine 81
6.1 Introduction . . . 82
6.2 Related work for Twine . . . 83
6.3 Twine design . . . 85
6.3.1 Requirements elicitation . . . 85
6.3.2 Threat model . . . 85
6.3.3 Architectural overview . . . 86

6.4 Implementation details . . . 88
6.4.1 Memory allocation . . . 89
6.4.2 Communication support . . . 90
6.4.3 File management support . . . 91
6.4.4 Attestation . . . 92
6.5 Demonstrating Twine with a distributed financial technology . . . 93
6.6 Evaluation . . . 95
6.6.1 Experimental setup . . . 95
6.6.2 Micro-benchmarks: PolyBench/C . . . 96
6.6.3 Micro-benchmarks: network stack . . . 98
6.6.4 Macro-benchmarks: SQLite . . . 100
6.6.5 Macro-benchmarks: Twine for credit scoring . . . 107
6.7 Security analysis . . . 108
6.8 Synthesis and next steps . . . 110

7 A trusted WebAssembly runtime for Arm TrustZone: WaTZ 111
7.1 Introduction . . . 112
7.2 Related work for WaTZ . . . 113
7.3 WaTZ: system overview . . . 114
7.3.1 Threat model . . . 114
7.3.2 Design overview . . . 115
7.3.3 Embedded trusted runtime . . . 116
7.4 Remote attestation of WebAssembly . . . 117
7.4.1 WaTZ protocol for remote attestation . . . 118
7.5 Implementation . . . 120
7.5.1 The runtime (attester) . . . 121
7.5.2 The server (verifier) . . . 122
7.5.3 The attestation service . . . 122
7.5.4 Extension to WASI: WASI-RA . . . 123
7.6 Evaluation . . . 124
7.6.1 Time measurements in TrustZone . . . 125
7.6.2 Startup overhead . . . 125
7.6.3 Micro-benchmarks: PolyBench/C . . . 127
7.6.4 Macro-benchmarks: SQLite . . . 128
7.6.5 Micro-benchmarks: remote attestation . . . 129
7.6.6 Macro-benchmarks: remote attestation with Genann . . . 131
7.7 Security analysis . . . 133
7.8 Synthesis and next steps . . . 134

III Establishing trust using WebAssembly for network communication 135

8 Trustworthy distributed systems with WebAssembly and TEEs 137
8.1 Introduction . . . 138

8.2 Publish/subscribe systems . . . 139
8.2.1 Publish/subscribe with TEEs . . . 140
8.3 Communication channel and attestation binding . . . 141
8.3.1 Binding communication protocols and attestation using TEEs . . . 142
8.4 Design considerations . . . 143
8.4.1 Threat model . . . 143
8.4.2 Security requirements . . . 144
8.4.3 Trusted primitives . . . 144
8.5 Architecture . . . 145
8.5.1 Attesting communication channels . . . 146
8.5.2 Securing publish/subscribe systems . . . 147
8.5.3 Implementation . . . 148
8.6 Evaluation . . . 150
8.6.1 Establishing new connections . . . 150
8.6.2 Messages throughput . . . 151
8.6.3 Scaling the publishers . . . 152
8.7 Synthesis . . . 153

9 Conclusion **155**
9.1 Thesis summary and takeaways . . . 156
9.2 Research perspectives and future directions . . . 158
9.2.1 Better WebAssembly compatibility with legacy applications . . . 158
9.2.2 Attestable and offloadable computations . . . 159
9.2.3 Widening support for emerging trusted execution environments . . . 160
9.2.4 Standardisation of attestation primitives and protocols . . . 161
9.2.5 Live migration of WebAssembly applications . . . 161
9.3 Concluding remarks . . . 162

10 References **165**

List of acronyms

ABI	application binary interface
ACL	access control list
AEAD	authenticated encryption with associated data
AES	Advanced Encryption Standard
ANN	artificial neural networks
AOT	ahead-of-time compilation
API	application programming interface
ASID	address space identifier
TA	trusted application
BFT	Byzantine fault-tolerant
CA	certificate authority
CAAM	NXP Cryptographic Accelerator and Assurance Module
CBC	cipher block chaining
CCA	Arm Confidential Compute Architecture
CCC	Confidential Computing Consortium
CCM	counter with cipher block chaining message authentication code
CFB	cipher feedback
CIL	Microsoft Common Intermediate Language
CLR	Microsoft Common Language Runtime
CMAC	cipher-based message authentication code
CoVE	RISC-V Confidential VM Extension
CPU	central processing unit
DAA	direct anonymous attestation
DCAP	Intel Data Center Attestation Primitives
DNN	deep neural network
DNS	Domain Name System
ECALL	Intel SGX enclave call
ECC	elliptic-curve cryptography
ECDH	elliptic curve Diffie-Hellman
ECDHE	elliptic curve Diffie-Hellman ephemeral
ECDSA	elliptic curve digital signature algorithm
ECH	encrypted client hello
EDMM	Intel SGX Enclave Dynamic Memory Management
ELF	Executable and Linkable Format
EPC	Intel SGX Enclave Page Cache
EPCM	Intel SGX Enclave Page Cache Map
EPID	Intel Enhanced Privacy ID
FaaS	function as a service
FHE	fully homomorphic encryption
GCM	Galois/Counter Mode of Operation
GP	GlobalPlatform
HSM	hardware security module
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IAS	Intel Attestation Service
IETF	Internet Engineering Task Force
IOMMU	input/output memory management unit
IoT	Internet of things
IPFS	Intel Protected File System
IR	intermediate representation
ISA	instruction set architecture
JIT	just-in-time compilation
JVM	Java virtual machine
KDK	key derivation key
KMS	key management system
LRU	least recently used
MAC	message authentication code
MEE	memory encryption engine
MIG	multi-instance GPU
MIME	Multipurpose Internet Mail Extensions
MKVB	master key verification blob
MMU	memory management unit
MPU	memory protection unit
NaCl	Google Native Client
NS	Arm TrustZone non-secure bit

OCALL Intel SGX outside call
OS operating system
OTPMK one-time programmable master key
PCC proof-carrying code
PCE Intel Provisioning Certification Enclave
PCK Intel Provisioning Certification Key
PCL Intel SGX protected code loader
PEF IBM OpenPOWER Protected Execution Facility
PKI public key infrastructure
PMP RISC-V Physical Memory Protection
PNaCl Portable Native Client
POSIX Portable Operating System Interface
PRNG pseudorandom number generator
PSK pre-shared key
PSW Intel SGX Platform Software
PUF physical unclonable function
QE Intel Quoting Enclave
QoS quality of service
RATS Remote Attestation Procedures
RMM Arm Realm Management Monitor
RMP AMD Reverse Map Table
RPC remote procedure call
RTL register transfer language
SDK software development kit
SEAM Intel Secure-Arbitration Mode
SEV AMD Secure Encrypted Virtualization
SEV-ES AMD SEV Encrypted State
SEV-SNP AMD SEV Secure Nested Paging
SFI software fault isolation
SGX Intel Software Guard Extensions
SLOC source lines of code
SMC Arm Secure Monitor Call
SME AMD Secure Memory Encryption
S/MIME Secure/Multipurpose Internet Mail Extensions
SMPC secure multi-party computation
SMTP Simple Mail Transfer Protocol
SoC system-on-chip
AMD-SP AMD Secure Processor
SVM IBM OpenPOWER PEF secure virtual machine
TC trusted computing
TCB trusted computing base
TD Intel Trust Domain
TDX Intel Trust Domain Extensions
TEE trusted execution environment
TF-A Trusted Firmware-A
TF-M Trusted Firmware-M
TLB process translation lookaside buffer
TLS Transport Layer Security
TME Intel Total Memory Encryption
TPM Trusted Platform Module
TSM RISC-V CoVE TEE security manager
TVM RISC-V CoVE trusted virtual machine
TZASC Arm TrustZone Address Space Controller
TZMA Arm TrustZone Memory Adapter
TZPC Arm TrustZone Protection Controller
UUID universally unique identifier
VFS virtual file system
VM virtual machine
VCEK AMD Versioned Chip Endorsement Key
VMCB AMD Virtual Machine Control Block
W3C World Wide Web Consortium
WABT WebAssembly Binary Toolkit
WAMR WebAssembly Micro Runtime
WASI WebAssembly System Interface
Wasm WebAssembly
WAT WebAssembly textual format
XEX xor-encrypt-xor
XTS XEX tweakable block cipher with ciphertext stealing

List of figures

2.1 The trust boundaries of the legacy computing stack and TEE implementations . . . 21
2.2 The execution flow of running a trusted application using Intel SGX 26
2.3 The execution flow of running a trusted application using Arm TrustZone 29
2.4 The execution flow of running a virtual machine using AMD SEV 32

3.1 The RATS architecture 41
3.2 The topological patterns of the RATS architecture 43
3.3 The target and attesting environments of the RATS architecture 44
3.4 The local attestation flow of Intel SGX 46
3.5 The remote attestation flow of Intel SGX EPID 47
3.6 The remote attestation flow of Intel SGX DCAP 49
3.7 The remote attestation flow of AMD SEV and AMD SEV-ES 51
3.8 The remote attestation flow of AMD SEV-SNP 52

4.1 An abstract representation of a Wasm runtime 59
4.2 The usage of a Wasm runtime within the three primary TEE trust boundaries . . . 64

5.1 The workflow of sending a secure message using IncaMail 76
5.2 The workflow of receiving a secure message using IncaMail 76
5.3 The speedup of the Wasm encryption and decryption operations 78
5.4 The scalability of the encryption cipher using Wasm 78

6.1 The Twine architecture 86
6.2 The deployment and attestation workflow of Twine 92
6.3 The architectural overview and workflow of Credora 94
6.4 The performance of PolyBench/C benchmarks 96
6.5 The performance of WolfSSL benchmarks targeting cryptographic algorithms . . . 98
6.6 The performance of WolfSSL benchmarks for TLS sessions. 99
6.7 The relative run time of SQLite Speedtest1 benchmarks 100
6.8 The performance of SQLite micro-benchmarks 102
6.9 The comparison of normalised run time for SGX variants 105
6.10 The run time breakdown prior to and following the optimisations of IPFS 106
6.11 The pulling time of Credora private pullers using SGX and Twine 107

7.1 The architecture of WaTZ 115
7.2 The architecture of the prototype 120
7.3 The time retrieval and world transition latencies 125
7.4 The startup breakdown of Wasm applications in WaTZ 126
7.5 The performance of PolyBench/C 127
7.6 The performance of SQLite's Speedtest1 128
7.7 The execution time of WaTZ's message 3 129

7.8 The execution time of the training phase 132

8.1 The architecture of the proposal 145
8.2 The enhanced TLS 1.3 handshake with attestation 146
8.3 The latency for each new connection 151
8.4 The system's throughput by measuring the latency of delivering messages 152

List of tables

2.1 The comparison of the state-of-the-art industrial and academic TEEs 36

4.1 The comparison of the state-of-the-art Wasm standalone runtimes 67

6.1 The comparison of the technologies in normalised run time 103
6.2 The cost factors of the micro-benchmarks 104

7.1 The comparison of the related work features 114
7.2 The remote attestation protocol of WaTZ 119
7.3 The execution time of WaTZ's messages 0, 1 and 2 130
7.4 The execution time breakdown of the WASI-RA API 132

8.1 The comparison of the state-of-the-art pub/sub systems shielded by TEEs 141
8.2 The comparison of the state-of-the-art channel binding solutions 143

Chapter 1

Introduction

This chapter explores the landscape of cloud, edge, and confidential computing, addressing security challenges and the need for more practical privacy-preserving solutions. The introduction highlights the key scientific contributions of the thesis, the open-source software contributions, the overall structure of the manuscript, and the related publications.

Chapter outline

1.1 The landscape of cloud, edge and confidential computing . . . 2
1.2 A brief history of portable compilation targets . . . 4
1.3 Research challenges . . . 5
1.3.1 Abstracting the complexity of multi-paradigms TEEs . . . 5
1.3.2 Attestation and communication between mistrusted parties . . . 7
1.4 Thesis contributions . . . 9
1.5 Thesis outline . . . 10
1.6 Attribution of contributions . . . 11
1.7 Published papers . . . 12
1.8 Open-source software contributions . . . 14

1.1 The landscape of cloud, edge and confidential computing

Over the last decades, computer systems have undergone significant changes. Originally, large-scale mainframe computers formed the backbone of computing infrastructure. However, the increasing power and affordability of smaller servers, alongside the demand for more flexible and scalable IT infrastructure and the growth of network technologies that facilitate efficient communication between distributed systems, led to a shift towards multi-server architectures. Concurrently, modern software paradigms promote abstractions from the underlying hardware and operating systems, enabling companies to offer services for code execution and data storage, commonly referred to as *cloud computing* [42]. As a result, cloud computing stands as one of the primary infrastructures supporting applications in the present day.

In addition to the rise of cloud computing, miniaturisation, which involves reducing the size of electronic components to create smaller, more compact chips, has led to the development of more efficient and powerful small-scale devices. These devices are collectively referred to as *edge computing* and have recently gained significant traction [43]. Edge computing refers to the enabling technologies that allow computation to be distributed at the edge of the network, on downstream data on behalf of cloud services, and upstream data on behalf of Internet of things (IoT) services. Specifically, the *edge* includes any computing and network resources along the path between data sources and cloud data centres. Similarly to cloud computing, edge computing is driven by the increasing demand for real-time data processing, reduced latency, and improved efficiency. As such, the growth of connected devices and the need to process data closer to the source, rather than transmitting it to centralised servers or cloud infrastructure, have fuelled the adoption of edge computing solutions.

The shift towards cloud and edge computing has introduced new security challenges for both customers and providers [44, 45]. From a customer's perspective, the loss of physical control over hardware and the potential for colocation with other tenants on shared infrastructure raise concerns about data confidentiality and the integrity of the execution environment. Customers must trust that the cloud provider will adequately isolate their workloads and protect sensitive information from unauthorised access or tampering by other tenants or the provider itself [46, 47]. Conversely, cloud providers face the challenge of securely executing untrusted customer software, which may contain malicious behaviour targeting the provider's infrastructure [48–51] or other customers' applications [52–54]. Providers must implement robust isolation mechanisms and security controls to prevent such attacks while still offering the flexibility and scalability that customers demand. Balancing these competing security requirements is a complex task that requires careful consideration of the trust relationships between customers, providers, and the underlying hardware and software stack.

The security challenges introduced by the shift towards cloud and edge computing have highlighted the need for privacy-preserving technologies. These technologies aim to enable secure and confidential computation of sensitive data, addressing the concerns of both customers and providers in the cloud and edge computing paradigms. Various technologies have been proposed to enable privacy-preserving computations, each with its own trade-offs and limitations. Fully homomorphic encryption (FHE), for example, allows computation on encrypted data without decryption but suffers from significant computational overheads that constrain its practical

adoption [55–57]. Similarly, secure multi-party computation (SMPC) enables multiple parties to collaborate on a computational task while keeping their individual inputs private [58]. However, it incurs substantial network overheads due to the need for extensive communication between the parties [59, 60]. In addition to those software-based approaches, trusted computing (TC) with Trusted Platform Modules (TPMs) provides a hardware-based solution for secure key storage, cryptographic operations, and attestation. However, TC does not support arbitrary mathematical operations or code execution, limiting its applicability in privacy-preserving computations [61]. While FHE, SMPC, and TC offer strong security guarantees, their performance or versatility limitations have motivated the exploration of alternative approaches to privacy-preserving computations that can better balance security, efficiency, and versatility.

Confidential computing has emerged as a promising approach to address the security challenges in cloud and edge computing environments. It aims to establish a secure, privacy-preserving computing paradigm where cloud providers and their customers can agree on common security grounds. Confidential computing aims to protect sensitive data and code from unauthorised access, tampering, or disclosure, even in the presence of untrusted or compromised infrastructure. To achieve this, confidential computing leverages trusted execution environments (TEEs), which are secure, isolated environments provided by most modern CPU manufacturers to execute code with confidentiality and integrity guarantees. TEEs root trust in the processor itself, offering hardware-based isolation and attestation mechanisms. Attestation provides proof of the integrity and authenticity of the TEE, enabling local and remote verification of the trusted environment. These mechanisms reduce the attack surface to the CPU, ensuring that sensitive code and data remain protected even if the underlying operating system or hypervisor is compromised. While TEEs rely on trusting the CPU and its manufacturer, recent research has demonstrated that TEEs offer an efficient and secure way to execute code [62–65]. Hence, TEEs provide a strong foundation for building confidential computing solutions that protect customers' and cloud providers' privacy in an increasingly interconnected computing landscape.

Despite the security benefits TEEs offer, they also introduce new challenges and limitations that must be carefully considered. Developing applications for TEEs often requires adopting a different programming paradigm, as the secure environment restricts the available system resources and interfaces for security purposes [66]. This usually leads to increased complexity and development overhead, as developers must adapt their code to work within the constraints of the TEEs. Additionally, TEE development is often tied to specific toolchains and programming languages supported by the hardware vendor, limiting security, flexibility and portability [67, 68]. While TEEs have a robust threat model, the trusted applications remain at risk if developed without security considerations. Hence, developers must exercise caution to avoid inadvertently leaking sensitive information through I/O communication, software bugs, and side channels, such as timing attacks or memory access patterns [69–74]. The risk of such leaks is increased because debugging and testing TEE applications can be more challenging due to the restricted environment [75, 76]. This dissertation aims to investigate and propose novel approaches that ease the development of trusted applications, tackling key challenges inherent in designing and implementing secure software for TEEs.

1.2 A brief history of portable compilation targets

Portable compilation targets, although developed independently, have played a relevant role in the development of applications for TEEs by enabling researchers and developers to more easily create secure applications that can be executed within these environments [77–81].

The history of portable compilation targets dates back to the early days of computing. One of the earliest examples is the p-code system, introduced by N. Wirth in the 1970s as part of the Pascal programming language [82, 83]. P-code provided a machine-independent intermediate representation (IR), allowing Pascal programs to be compiled once and executed on various platforms. Inspired by academic research [84, 85], mainstream compilers also introduced IRs as an abstraction between high-level programming languages and platform-specific assembly code. A notable example of the early adoption of IR is the register transfer language (RTL) used in the GCC compiler [86], which enabled the compiler to translate high-level code into RTL before generating CPU-specific machine instructions. This concept of platform-independent code execution laid the foundation for future portable compilation targets.

In the 1990s, the Java programming language introduced the Java virtual machine (JVM), which became a widely adopted portable compilation target [87]. The JVM allowed Java bytecode to be executed on any platform that supported the virtual machine, providing a high level of portability. This approach was later extended to other languages, such as Microsoft's Common Language Runtime (CLR) for the .NET framework (and .NET, its successor) [88–90]. These virtual machines provided a sandboxed environment for code execution, offering a degree of security and isolation from the underlying system. Moreover, the JVM and CLR enabled running software on heterogeneous devices, from resource-constrained IoT devices to powerful server-grade machines, further enhancing their versatility and adoption over the last decade.

In the context of web technologies, Google introduced Native Client (NaCI) [91] in 2009 as a sandboxing technology for securely running native code in web browsers. NaCI aimed to improve the performance of web applications by allowing developers to leverage native code execution while maintaining the security guarantees of the web sandbox. NaCI initially supported a subset of x86, Arm, and MIPS instructions executed within the NaCI sandbox. Later, Google introduced Portable Native Client (PNaCl) [92], which used LLVM bytecode (called LLVM IR) as a portable intermediate representation. PNaCl allowed developers to compile their code once and run it on multiple architectures, providing better portability compared to NaCI. Similarly, asm.js [93], introduced in 2013, was a subset of JavaScript designed to serve as a low-level compilation target for languages like C and C++. Asm.js enabled near-native performance for web applications by providing a structured and optimizable subset of JavaScript that could be efficiently compiled and executed by web browsers without relying on a dedicated runtime.

WebAssembly (Wasm) [94], introduced in 2017, represents a large improvement in portable compilation targets. Developed as a World Wide Web Consortium (W3C) open standard by many renowned companies such as Microsoft and Google, Wasm is designed to be a lightweight, efficient, and secure platform for executing code across many environments. Despite its name, one of the key features of Wasm is its adaptability to both web and standalone runtimes. In web browsers, Wasm modules can be executed alongside JavaScript, providing a high-performance

alternative for computationally intensive tasks [95]. Standalone Wasm runtimes enable the execution of Wasm modules as native applications, independent of web browsers [96]. These runtimes use a standardised system interface called WebAssembly System Interface (WASI), which provides Wasm applications with controlled access to underlying system resources [97].

Wasm stands out from previous portable compilation targets through its modular design. Unlike technologies such as the JVM and CLR, which come with large built-in standard libraries and implementation requirements like garbage collection, Wasm adopts a modular approach. This approach involves an opt-in strategy for additional features, resulting in a minimal execution environment that is particularly well-suited for constrained environments [98, 99]. As such, Wasm does not define any standard library or require binding to specific programming components, such as a garbage collector. Although Java and .NET applications can be built without standard libraries, doing so is incompatible with most programs and libraries, as they heavily rely on that bloated infrastructure. Wasm changes this approach by proposing a minimal environment where programming toolchains deploy only the necessary components to run the programs. For instance, a program written in Rust can be compiled to Wasm without a standard library, resulting in a lightweight executable. On the other hand, high-level programming languages, such as Go, may compile their standard library and other software dependencies, like the garbage collector, into Wasm, leading to a more substantial binary size.

Moreover, Wasm adopts a capability-based security model [100], requiring code modules to declare and request access to specific system resources explicitly. Therefore, the Wasm runtime must grant access to OS or browser services, inverting the strategy of existing portable technologies that assume they inherit the same system permissions as the host runtime. A sandbox enforces Wasm's capability-based security model that restricts boundaries between the executing code and the host environment. This sandboxing is achieved through memory isolation, control flow restrictions [101], and a well-defined interface for interaction with the host [97]. The fine-grained control over permissions helps prevent unintended or malicious access to sensitive API or data. Given the unique modular approach and lightweight nature of Wasm, this dissertation explores how the combination of Wasm and TEEs can enhance the security, portability, and efficiency of confidential computing applications.

1.3 Research challenges

The rise of confidential computing has introduced numerous challenges that both academia and industry are actively tackling. This thesis focuses on the abstraction of programming trusted applications to target multiple TEEs using Wasm as a portable compilation target, aiming to address the portability and security challenges inherent in developing trusted software.

1.3.1 Abstracting the complexity of multi-paradigms TEEs

Today, processor manufacturers have their unique approaches to implementing their TEE architectures. Hence, TEEs come in various forms, such as process-based (Intel SGX [102], Sanctum [103]), partition-based (Arm TrustZone [104], Keystone [105]), and virtual machine (VM)-

based (AMD SEV [106], Intel TDX [107], Arm CCA [108]). Consequently, each TEE has its dedicated security threat model, development paradigm, and attestation process for establishing trust with the deployed software. This heterogeneity in TEE architectures presents a severe challenge for developers to effectively create and maintain portable and secure applications that seamlessly operate across different processor architectures.

Researchers have proposed various solutions to abstract the low-level implementation details of TEEs [62–64, 109]. These approaches, however, often target a single TEE at a time and emulate the complexity of Unix-like systems to varying degrees, such as by hooking system calls for internal handling. While these systems can protect unmodified Linux binaries inside TEEs, they create a strong coupling with the application binary interface (ABI), the Executable and Linkable Format (ELF), and the instruction set architecture (ISA) for which the trusted application has been compiled. This tight dependency on specific architectures and formats reduces the portability and adaptability of these solutions across different TEE implementations.

Moreover, moving portions of kernel code into the TEE often leads to a large trusted computing base (TCB), which increases the likelihood of security vulnerabilities, as the number of bugs tends to grow in proportion to code size [110]. The limited memory capacity of some TEEs further worsens the issue. For instance, Arm TrustZone, when using the trusted operating system OP-TEE [111], may provide as little as a few megabytes of memory, rendering the TEE non-functional when the TCB size exceeds the available memory [112]. Similarly, TEEs like Intel SGX can experience decreased performance due to their memory constraints [113]. Although prior work has attempted to reduce the TCB via code partitioning [80, 114, 115], these efforts still suffered from a tight coupling with a particular TEE or processor architecture, limiting their applicability and effectiveness across heterogeneous TEE environments.

In contrast to previous work, this dissertation aims to create a common ground for different TEEs with distinct ISAs by leveraging Wasm as a lightweight and platform-agnostic abstraction layer. Wasm is a prime candidate for introducing this abstraction, because it is an intermediate bytecode format that is designed to work efficiently on various platforms and processor architectures [96, 116]. Some state-of-the-art Wasm runtimes have a small binary size (under a megabyte) and are designed to run in constrained environments, such as TEEs [117, 118]. Trusted applications can be compiled from many programming languages to WebAssembly and are loosely coupled from the operating system, as they do not rely on platform-specific features or system calls. WASI enables the abstraction of the system interface, simplifying the process of intercepting I/O calls for securing interactions without relying on low-level system call hooking. For example, securing file system interaction requires providing only an alternate implementation for about ten (out of 46) WASI functions [119].

The adoption of Wasm as a common abstraction layer for TEEs offers a promising solution to the inherent distrust between cloud providers and their customers. By leveraging Wasm, this approach establishes a two-way sandbox that effectively addresses the security concerns of both parties [120]. From the customer's perspective, TEEs provide a trusted boundary that ensures the confidentiality and integrity of their sensitive data and computations. The hardware-enforced isolation mechanisms of TEEs shield the customer's software from unauthorised access or tampering by the cloud provider or other tenants sharing the same infrastructure. Conversely, the sandbox of Wasm protects the cloud provider's assets by constraining the behaviour of the

customer's software within well-defined boundaries. The Wasm runtime enforces strict security policies, preventing potentially malicious or buggy code from compromising the stability or security of the underlying platform, such as the OS or the runtime in the TEE.

The combination of TEEs and Wasm creates a mutually beneficial security model that promotes trust and collaboration in the confidential computing ecosystem. The two-way sandbox reduces the risk of customers exploiting vulnerabilities or mounting attacks against the cloud provider's infrastructure while also ensuring customer privacy. To establish trust, both parties must agree upon a reference implementation of the Wasm runtime, ensuring its security properties are enforced correctly and well-understood. The agreement of both parties on a particular implementation provides a common ground for deploying trusted applications, laying the foundation for a more transparent and trustworthy relationship between cloud providers and their customers.

This thesis demonstrates the performance and security advantages of using Wasm in web applications while tackling the challenges of porting a Wasm runtime to two distinct TEEs with different ISAs in both cloud and edge computing settings. This research investigates how to secure I/O operations within these environments, ensuring the confidentiality and integrity of data as it crosses the trust boundary, ultimately contributing to developing a common abstraction layer for confidential computing.

1.3.2 Attestation and communication between mistrusted parties

While the isolation offered by TEEs is essential for preserving the confidentiality and integrity of trusted applications and their associated data, a typical use case for TEEs involves receiving or transferring data to third-party entities [121]. The isolation guarantees provided by TEEs become truly meaningful and valuable only when coupled with the ability to verify the authenticity of a given TEE instance, including the software it executes. Without this verification, the security properties of TEEs cannot be fully leveraged, as there is no assurance that the TEE has not been tampered with or is running the intended software [122, 123].

Attestation is the process commonly used to establish trust in specific TEE instances [124]. This process generates a cryptographic proof comprising various metrics, such as a hash of the executing trusted application and a signature from the processor [125, 126]. The cryptographic materials used in attestation are generally fused into the processor's die during manufacturing, preventing attackers from forging malicious attestation proofs. When a TEE instance issues a proof, it is communicated to a third-party actor who appraises it to determine the genuineness of the TEE and ensures that sensitive data is only transferred to a secure and up-to-date environment. Hence, the attestation process establishes a trust relationship between the TEE and the external entity, which will serve as a foundation for securing future communication.

Attestation is, therefore, a fundamental feature for using TEEs. However, like TEEs themselves, attestation suffers from a lack of standardisation, leading processor manufacturers to provide ad-hoc solutions for attestation support [127, 128]. This lack of standardisation further complicates the development process for TEE-based applications, as the lack of a unified approach to attestation across different TEE implementations creates a fragmented landscape [129]. Moreover, attestation is a careful process that can lead to various attacks and abuses if not carried

out correctly [130]. As a result, developers must understand vendor-specific APIs and security considerations, increasing the likelihood of introducing vulnerabilities or misconfigurations.

While generating an attestation is a complex topic in itself, creating a secure communication channel between a third-party actor and a given TEE instance is also of utmost importance, as attestation and communication are closely related [131]. However, processor manufacturers often do not provide comprehensive, ready-to-use solutions for secure communication, leaving developers to rely on basic open-source examples or community-driven initiatives, such as with Intel's proposal [132]. Other open-source projects, such as Open Enclave [133], Gramine [63], and Inclavar Containers [134], have also attempted to address this gap by implementing variations of secure communication protocols [135–137]. However, these projects lack interoperability, leading to a fragmented landscape of secure communication solutions in TEEs. Developers are forced to choose between incompatible implementations, limiting the potential for building applications that seamlessly span multiple TEE platforms. This lack of standardised communication mechanisms further challenges developers building applications that span multiple TEEs, as they must invest considerable effort in implementing custom protocols to ensure the confidentiality and integrity of data in transit [138].

Academia has actively contributed building blocks for attesting and securely communicating with TEEs. Nonetheless, these solutions are often tied to specific TEE implementations [62, 121], communication protocols [131, 139–142], or development frameworks [135–137], limiting their broader applicability and interoperability in heterogeneous environments. The Confidential Computing Consortium (CCC) Attestation Special Interest Group has recently started efforts to develop modern, unified solutions for handling attestation independently of the underlying TEE implementation [143, 144]. These initiatives aim to provide a standard approach to attestation and establish trusted communication channels, enabling seamless interoperability across many TEE technologies. While promising, these solutions are still in the early stages of development and require further research to reach maturity and widespread adoption.

As a potential solution to TEE-neutral attestation, this thesis proposes decoupling trusted applications from the underlying development frameworks and TEE-specific attestation support by leveraging trusted Wasm runtimes. By adopting this approach, Wasm applications running in TEEs across different architectures can use a generic approach for generating, storing and appraising attestation proofs. This abstraction layer shifts the responsibility of handling the low-level primitives exposed by the processors to the trusted Wasm runtimes.

Furthermore, this dissertation expands the concept of a portable attestation mechanism, using it as a foundation to design a distributed and trusted communication system, enabling various TEE instances to mutually attest to each other and establish secure communication channels, regardless of the processor hardware vendors involved. The proposed interoperable solution for attestation and communication eases the development and deployment of TEE applications, supporting TEE instances running on different machines to communicate seamlessly without being tied to specific hardware vendors, programming languages, or operating systems.

1.4 Thesis contributions

In this thesis, I present a comprehensive approach to enabling the secure and efficient execution of applications within TEEs. The first topic focuses on the design and implementation of secure Wasm runtimes that leverage the features of TEEs, providing a foundation for confidential computing and ensuring the confidentiality and integrity of Wasm applications and their related data within these protected environments. Additionally, I provide novel mechanisms for establishing trust between mutually attested TEEs, enabling secure communication channels and easing the development of distributed applications that span multiple TEEs. The key contributions of this thesis are summarised in the following paragraphs.

Survey of TEEs and attestation

We reviewed remote attestation principles and compared how the modern trusted execution environments Intel SGX, Arm TrustZone and AMD SEV, as well as emerging RISC-V solutions, leverage these mechanisms [6, 7]. This survey highlights that existing solutions widely differ in maturity and security. Whereas some TEEs are developed by leading processor companies and provide built-in attestation mechanisms, others still lack proper hardware attestation support. This work serves as a foundation for designing and implementing trusted Wasm runtimes, secure attestation protocols, and communication mechanisms further developed in this dissertation.

Vision of cloud-edge continuum with Wasm

The transformation of the cloud computing landscape from a centralised architecture to a distributed and heterogeneous one, including edge and IoT devices, has given rise to the *cloud-edge continuum*, bridging the gap between data centres and end-user devices. Existing solutions for programming the continuum are, however, dominated by proprietary silos and incompatible technologies built around dedicated devices and runtime stacks. Throughout this thesis, we motivate the need for an interoperable environment that can run seamlessly across hardware devices and software stacks, while achieving good performance and a high level of security; a critical requirement when processing data off-premises. Our work demonstrates that the technology provided by Wasm running on modern virtual machines and shielded within TEEs, combined with a core set of services and support libraries, allows us to meet both goals [5].

Speed-up cryptography and reduce threat model using Wasm

JavaScript has traditionally been used for cryptographic operations in web clients when native API is unavailable or lacks support for required ciphers. However, the interpreted nature of JavaScript and its limited low-level optimisations lead to performance bottlenecks, especially for compute-intensive tasks such as encryption and decryption. Wasm offers a compelling alternative to improve the execution of cryptographic primitives in browsers. We demonstrate the practical benefits of client-side Wasm-based cryptography and show how it can significantly outperform conventional JavaScript implementations in the use case of Swiss Post's IncaMail [145], a secure email service [3]. Furthermore, Wasm is used to reduce the TCB of the IncaMail, enabling an honest-but-curious model without compromising message confidentiality.

A trusted Wasm runtime for Intel SGX

We present TWINE [8], a trusted runtime for running Wasm-compiled applications within TEEs, establishing a two-way sandbox. TWINE leverages memory safety guarantees of Wasm and abstracts the complexity of TEEs, empowering the execution of legacy and language-agnostic applications. It extends the standard WASI, providing controlled OS services, focusing on I/O. Additionally, through built-in TEE mechanisms, TWINE offers an attestation mechanism to ensure the integrity of the runtime and the OS services supplied to the application. We evaluate its performance using general-purpose benchmarks and real-world applications, showing it matches state-of-the-art solutions. A case study involving fintech company *Credora* reveals that TWINE can be deployed in production with reasonable performance trade-offs. Finally, we identify performance improvement through library optimisation, showcasing one such adjustment.

A trusted Wasm runtime for Arm TrustZone

We present WATZ [4], an efficient and secure runtime for trusted execution of Wasm code for Arm's TrustZone TEE. WATZ also includes a lightweight remote attestation system optimised for Wasm applications running in TrustZone, addressing the lack of built-in attestation mechanisms. The remote attestation protocol is formally verified using a state-of-the-art analyser and model checker. Our evaluation of Arm-based hardware uses synthetic and real-world benchmarks, illustrating typical tasks IoT devices achieve. WATZ's execution speed is on par with Wasm runtimes in the normal world and reaches roughly half the speed of native execution, which is compensated by the additional security guarantees and the interoperability offered by Wasm.

Secure communication between TEEs

We introduce the design of a portable and fully attested publish/subscribe (pub/sub) middleware system as a holistic approach for trustworthy and distributed communication between various systems [2]. Pub/sub systems play a key role in enabling communication between numerous devices in distributed and large-scale architectures. Based on this proposal, we have implemented and evaluated a pub/sub broker running within Intel SGX, compiled in Wasm, and built on top of industry-battled frameworks and standards, *i.e.*, MQTT and TLS protocols. Our extended TLS protocol preserves the privacy of attestation information, among other benefits.

1.5 Thesis outline

This thesis is organised into nine chapters, presenting the research findings in a logical sequence for the reader's convenience rather than adhering to a strict chronological order.

Part I introduces the core concepts and technologies further developed in this manuscript. Chapter 2 provides an overview of the TEEs, whereas Chapter 3 explains attestation and its utmost importance in establishing trust in TEEs. Chapter 4 introduces Wasm, a compact binary format and portable compilation target, which is designed for secure application execution in browsers and standalone runtimes. Part II explores how Wasm can serve as an efficient and lightweight abstraction layer for TEEs. Chapter 5 evaluates the performance of Wasm in comparison to

traditional web languages and how this technology can be used to reduce the threat model of web applications. The trusted runtimes TWINE and WATZ, which focus on securing application execution in TEEs using Intel SGX and Arm TrustZone, respectively, are presented in Chapters 6 and 7. Part III addresses the complexity of establishing trust for network communications using TEEs. In Chapter 8, we propose a publish/subscribe system that establishes secure communication channels thanks to mutual attestation, which is independent of the underlying TEE implementation, by leveraging Wasm as an abstraction mechanism. Chapter 9 concludes the dissertation by summarising the key scientific contributions, discussing their implications and exploring future research directions in the fields of Wasm and TEEs.

The remainder of this chapter outlines the scientific and open-source software contributions that emerged from the research conducted within the scope of this doctoral thesis.

1.6 Attribution of contributions

This dissertation builds upon collaborations with students, insightful researchers, and various research institutes. The scientific foundation of this thesis is derived from the resulting publications, which were developed with the invaluable support of the respective co-authors. In this section, I acknowledge the contributions of the co-authors by attributing the relevant parts of this dissertation to their corresponding ideas. Furthermore, I map each chapter of this dissertation to the underlying publications that we have authored.

Trusted execution environments The Computer Science Department of the University of Neuchâtel has been actively engaged in the security research field of TEEs. As such, this thesis builds upon prior work from team members [112, 146–148] and leverages these secure environments as the foundation of my research, using them as a mechanism to efficiently execute trustworthy applications on untrusted systems. In particular, my supervisors P. Felber, M. Pasin, and colleague V. Schiavoni proposed easing the usage of TEEs on various hardware platforms.

Attestation in confidential computing P. Felber, M. Pasin, and V. Schiavoni jointly proposed to examine the state-of-the-art techniques for attestation of TEEs. This joint effort led to a preliminary workshop publication [7], where C. Göttel provided insights into the internal workings of AMD SEV and Intel SGX. Later, we collaborated with RISE Research Institutes of Sweden to further develop our work. A. Khurshid made contributions by enhancing the analysis of Arm TrustZone-M. This expanded research has been published separately in a conference [6].

WebAssembly P. Felber and M. Pasin envisioned using Wasm as an abstraction layer for building secure and performant software that can run in TEEs regardless of the underlying hardware [5]. This vision was inspired by the efficient execution of this intermediate language [94] and the work of D. Goltzsche, R. Kapitza *et al*. in the context of TEEs [120].

A use-case of WebAssembly: IncaMail P. Felber, V. Schiavoni, and the Swiss Post engaged in discussions about the potential of leveraging Wasm to optimise the encryption performance of IncaMail. We further refined the design to offload cryptographic operations to client devices while strengthening the threat model [3]. As part of his master's thesis, P. Gerig developed the proof-of-concept implementation and evaluated the performance improvements.

A trusted WebAssembly runtime for Intel SGX: Twine The development of a trusted runtime for executing Wasm on Intel processors was influenced by prior work on trusted software abstractions [62–64]. We built upon the open-source WebAssembly Micro Runtime (WAMR) project, which initially offered limited support for Intel SGX. Our contributions resulted in a conference publication and multiple pull requests on the project repository [8]. Later, we collaborated with *Credora*, which provided funding, and G. Mazzeo from Parthenope University of Naples to further improve the trusted runtime and integrate it into *Credora*'s toolchain. The resulting work has been published in a journal and led to several pull requests (listed in §1.8) [1].

A trusted WebAssembly runtime for Arm TrustZone: WaTZ Continuing our efforts to abstract hardware using Wasm, we focused on developing a trusted runtime specifically designed for Arm TrustZone with remote attestation. The attestation mechanism was influenced by state-of-the-art protocols from both industry [122, 149] and academia [127], as well as emerging standards like the Remote Attestation Procedures (RATS) [150]. We further extended WAMR to serve as a trusted runtime and published our findings in a conference paper [4].

Trustworthy distributed systems with WebAssembly and TEEs We introduced a standardised communication scheme to fully exploit the potential of TEEs. Our proposed architecture leverages a pub/sub approach, enabling trusted parties to communicate while exchanging attestation proofs. As part of his master's thesis, J. Oeftiger modified the Transport Layer Security (TLS) protocol to include attestation information in the protocol's handshake. Additionally, A. Grüter focused his master's thesis on porting a pub/sub library to be compiled into Wasm. A. Grüter integrated J. Oeftiger's work into a proof-of-concept implementation, and we evaluated the performance overheads. The resulting work led to a conference publication [2].

This doctoral thesis incorporates results from the VEDLIoT project, which received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement n° 957197. Besides, *Credora* financed the research work extending Twine.

1.7 Published papers

The following peer-reviewed publications constitute the core of this manuscript, with each article either comprising an entire chapter or a significant portion thereof.[1]

[1]The numerical identifiers preceding the entries in the subsequent sections correspond to their respective bibliographic records within the reference list located at the end of the manuscript.

[1] **Jämes Ménétrey**, Marcelo Pasin, Pascal Felber, Valerio Schiavoni, Giovanni Mazzeo, Arne Hollum, and Darshan Vaydia. 2024. A comprehensive trusted runtime for WebAssembly with Intel SGX. *IEEE Transactions on Dependable and Secure Computing*, 21, 4, 3562–3579. DOI: 10.1109/TDSC.2023.3334516.

[2] **Jämes Ménétrey**, Aeneas Grüter, Peterson Yuhala, Julius Oeftiger, Pascal Felber, Marcelo Pasin, and Valerio Schiavoni. 2023. A holistic approach for trustworthy distributed systems with WebAssembly and TEEs. In *27th International Conference on Principles of Distributed Systems, OPODIS 2023, December 6-8, 2023, Tokyo, Japan* (LIPIcs). Vol. 286. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 23:1–23:23. DOI: 10.4230/LIPICS.OPODIS.2023.23.

[3] Pascal Gerig, **Jämes Ménétrey**, Baptiste Lanoix, Florian Stoller, Pascal Felber, Marcelo Pasin, and Valerio Schiavoni. 2023. Preventing EFail attacks with client-side WebAssembly: the case of Swiss Post's IncaMail. In *Proceedings of the 17th ACM International Conference on Distributed and Event-based Systems, DEBS 2023, Neuchatel, Switzerland, June 27-30, 2023*. ACM, 151–156. DOI: 10.1145/3583678.3596899.

[4] **Jämes Ménétrey**, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. 2022. WaTZ: a trusted WebAssembly runtime environment with remote attestation for TrustZone. In *42nd IEEE International Conference on Distributed Computing Systems, ICDCS 2022, Bologna, Italy, July 10-13, 2022*. IEEE, 1177–1189. DOI: 10.1109/ICDCS54860.2022.00116.

[5] **Jämes Ménétrey**, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. 2022. WebAssembly as a common layer for the cloud-edge continuum. In *FRAME@HPDC 2022: Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge, Minneapolis, MN, USA, 1 July 2022*. ACM, 3–8. DOI: 10.1145/3526059.3533618.

[6] **Jämes Ménétrey**, Christian Göttel, Anum Khurshid, Marcelo Pasin, Pascal Felber, Valerio Schiavoni, and Shahid Raza. 2022. Attestation mechanisms for trusted execution environments demystified. In *Distributed Applications and Interoperable Systems: 22nd IFIP WG 6.1 International Conference, DAIS 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings* (Lecture Notes in Computer Science). Vol. 13272. Springer, 95–113. DOI: 10.1007/978-3-031-16092-9_7.

[7] **Jämes Ménétrey**, Christian Göttel, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. 2022. An exploratory study of attestation mechanisms for trusted execution environments. In *5th Workshop on System Software for Trusted Execution, SysTEX 2022, co-located with ASPLOS'22, Lausanne, Switzerland, March 1, 2022*. https://systex22.github.io/papers/systex22-final79.pdf.

[8] **Jämes Ménétrey**, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. 2021. Twine: an embedded trusted runtime for WebAssembly. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, 205–216. DOI: 10.1109/ICDE51399.2021.00025.

The following peer-reviewed publications, although not forming the core of this dissertation, have been published during my doctoral studies and have contributed to contextualising the research presented herein.

[9] Louis Vialar, **Jämes Ménétrey**, Valerio Schiavoni, and Pascal Felber. 2024. BlindexTEE: a blind index approach towards TEE-supported end-to-end encrypted DBMS. In *Stabilization, Safety, and Security of Distributed Systems - 26th International Symposium, SSS 2024, Nagoya, Aichi, Japan, October 20-22, 2024, Proceedings* (Lecture Notes in Computer Science). Springer.

[10] Mpoki Mwaisela, Joel Hari, Peterson Yuhala, **Jämes Ménétrey**, Pascal Felber, and Valerio Schiavoni. 2024. Evaluating the potential of in-memory processing to accelerate homomorphic encryption. In *43rd International Symposium on Reliable Distributed Systems, SRDS 2024, Charlotte, USA, September 30–October 03, 2023*. IEEE.

[11] Peterson Yuhala, **Jämes Ménétrey**, Pascal Felber, Marcelo Pasin, and Valerio Schiavoni. 2024. Fortress: securing IoT peripherals with trusted execution environments. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing, SAC 2024, Avila, Spain, April 8-12, 2024*. ACM, 243–250. DOI: 10.1145/3605098.3635994.

[12] Kevin Mika, René Griessl, Nils Kucza, Florian Porrmann, Martin Kaiser, Lennart Tigges, Jens Hagemeyer, Pedro Trancoso, Muhammad Waqar Azhar, Fareed Qararyah, Stavroula Zouzoula, **Jämes Ménétrey**, Marcelo Pasin, Pascal Felber, Carina Marcus, Oliver Brunnegård, Olof Eriksson, Hans Salomonsson, Daniel Ödman, Andreas Ask, António Casimiro, Alysson Bessani, Tiago Carvalho, Karol Gugala, Piotr Zierhoffer, Grzegorz Latosinski, Marco Tassemeier, Mario Porrmann, Hans-Martin Heyn, Eric Knauss, Yufei Mao, and Franz Meierhöfer. 2023. VEDLIoT: next generation accelerated AIoT systems and applications. In *Proceedings of the 20th ACM International Conference on Computing Frontiers, CF 2023, Bologna, Italy, May 9-11, 2023*. ACM, 291–296. DOI: 10.1145/3587135.3592175.

[13] Martin Kaiser, René Griessl, Nils Kucza, Carola Haumann, Lennart Tigges, Kevin Mika, Jens Hagemeyer, Florian Porrmann, Ulrich Rückert, Micha vor dem Berge, Stefan Krupop, Mario Porrmann, Marco Tassemeier, Pedro Trancoso, Fareed Qararyah, Stavroula Zouzoula, António Casimiro, Alysson Neves Bessani, José Cecílio, Stefan Andersson, Oliver Brunnegård, Olof Eriksson, Roland Weiss, Franz Meierhöfer, Hans Salomonsson, Elaheh Malekzadeh, Daniel Ödman, Anum Khurshid, Pascal Felber, Marcelo Pasin, Valerio Schiavoni, **Jämes Ménétrey**, Karol Gugala, Piotr Zierhoffer, Eric Knauss, and Hans-Martin Heyn. 2022. VEDLIoT: very efficient deep learning in IoT. In *2022 Design, Automation & Test in Europe Conference & Exhibition, DATE 2022, Antwerp, Belgium, March 14-23, 2022*. IEEE, 963–968. DOI: 10.23919/DATE54114.2022.9774653.

[14] Peterson Yuhala, **Jämes Ménétrey**, Pascal Felber, Valerio Schiavoni, Alain Tchana, Gaël Thomas, Hugo Guiroux, and Jean-Pierre Lozi. 2021. Montsalvat: Intel SGX shielding for GraalVM native images. In *Middleware '21: 22nd International Middleware Conference, Québec City, Canada, December 6-10, 2021*. ACM, 352–364. DOI: 10.1145/3464298.3493406.

1.8 Open-source software contributions

This thesis aimed to contribute to the open-source communities that initially developed software used throughout the research in this dissertation. These contributions led to receiving the following prestigious award and recognition. The remaining of this section enumerates the contributions that have led to these acknowledgements.

- **HiPEAC Tech Transfer Award (2022)** for integrating Wasm with Intel SGX, specifically the award *Full-stack trusted WebAssembly runtime using Intel SGX enclaves for secure cryptocurrency credit scoring* [151].
- **Recognised Contributor** from the Bytecode Alliance for the numerous open-source contributions to the WAMR Wasm runtime [152].

Scientific reproducibility is a fundamental principle of rigorous research, ensuring that research findings can be independently verified and built upon by other researchers. Throughout this thesis, the work conducted has resulted in the creation of open-source software artefacts that encourage reproducible experiments. These artefacts are publicly available on GitHub, allowing other researchers to access, examine, and build upon the work presented in this thesis. The references to the relevant GitHub repositories are provided below.

[15] [SW] **Jämes Ménétrey**, TWINE runtime and experiments Jan. 15, 2024. URL: https://github.com/jamesmenetrey/unine-twine.

[16] [SW] **Jämes Ménétrey**, Aeneas Grüter, and Julius Oeftiger, A Holistic Approach for Trustworthy Distributed Systems with WebAssembly and TEEs: code and benchmarks Dec. 1, 2023. URL: https://github.com/JamesMenetrey/unine-opodis2023.

[17] [SW] **Jämes Ménétrey**, WATZ runtime and experiments May 19, 2022. URL: https://github.com/jamesmenetrey/unine-watz.

In addition to the software artefacts related to the published papers, software contributions have been made to upstream enhancements and bug fixes of open-source projects used in the research of this thesis. These contributions were submitted in the form of GitHub pull requests. The links to the relevant pull requests are also provided below.

Contributions to WAMR

[18] [SW] **Jämes Ménétrey**, Clarify how to verify SGX evidence without an SGX-enabled platform Feb. 17, 2024. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/3158.

[19] [SW] **Jämes Ménétrey**, Attestation: free JSON from the Wasm module heap Nov. 22, 2023. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/2803.

[20] [SW] **Jämes Ménétrey**, SGX-RA: disable the building of samples Aug. 28, 2023. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/2507.

[21] [SW] **Jämes Ménétrey**, Upgrade SGX-RA integration for 0.1.2 and Ubuntu 20.04 Aug. 15, 2023. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/2454.

[22] [SW] **Jämes Ménétrey**, Remove a file test outside of the specs and improve CI reporting Mar. 24, 2023. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/2057.

[23] [SW] **Jämes Ménétrey**, SGX IPFS: fix a segfault and support seeking beyond the end of files while using SEEK_CUR/SEEK_END Jan. 30, 2023. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1916.

[24] [SW] **Jämes Ménétrey**, linux-sgx: open files with any paths in the sandbox using IPFS Nov. 7, 2022. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1685.

[25] [SW] **Jämes Ménétrey**, linux-sgx: improve the documentation of SGX-RA Nov. 4, 2022. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1679.

[26] [SW] **Jämes Ménétrey**, linux-sgx: use non-destructive modes for opening files using SGX IPFS Oct. 27, 2022. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1645.

[27] [SW] **Jämes Ménétrey**, Normalize how the global heap size is defined across iwasm apps Oct. 25, 2022. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1628.

[28] [SW] **Jämes Ménétrey**, linux-sgx: implement POSIX calls based on getsockname and booloption Oct. 11, 2022. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1574.

[29] [SW] **Jämes Ménétrey**, linux-sgx: implement getpeername, recvfrom and sendto Oct. 6, 2022. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1556.

[30] [SW] **Jämes Ménétrey**, linux-sgx: fix directional OCALL parameter for getsockname Oct. 4, 2022. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1554.

[31] [SW] **Jämes Ménétrey**, Hash map: Fix a wrongly named parameter and enhance the docs Sept. 29, 2022. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1540.

[32] [SW] **Jämes Ménétrey**, Socket: Explicit narrowing type cast and add missing static keywords Sept. 29, 2022. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1539.

[33] [SW] **Jämes Ménétrey**, linux-sgx: Implement SGX IPFS as POSIX backend for files interaction Sept. 28, 2022. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1489.

[34] [SW] **Jämes Ménétrey**, Implement Berkeley Socket API for Intel SGX Mar. 25, 2022. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1061.

Contributions to Librats

[36] [SW] **Jämes Ménétrey**, sgx_ecdsa: set the load_policy at most once Nov. 17, 2023. URL: https://github.com/inclavare-containers/librats/pull/94.

[37] [SW] **Jämes Ménétrey**, verifiers: use type instead of name for selection Aug. 14, 2023. URL: https://github.com/inclavare-containers/librats/pull/82.

Contributions to WolfSSL

[38] [SW] **Jämes Ménétrey**, Implement a TLS perf server with poll system call Sept. 27, 2022. URL: https://github.com/wolfSSL/wolfssl-examples/pull/330.

[39] [SW] **Jämes Ménétrey**, TLS client: fix bad comparison for non-blocking shutdown call July 21, 2022. URL: https://github.com/wolfSSL/wolfssl-examples/pull/327.

Contributions to WebAssembly specifications

[40] [SW] **Jämes Ménétrey**, Update latex .bib file for citing WebAssembly specs 2.0 May 6, 2022. URL: https://github.com/WebAssembly/spec/pull/1463.

Contributions to OP-TEE

[41] [SW] **Jämes Ménétrey**, imx: add a compilation flag to support the persistence of the rootfs May 3, 2021. URL: https://github.com/OP-TEE/build/pull/477.

The following part of this dissertation covers the core concepts that support the research presented herein. In particular, the next chapter introduces TEEs in detail, explaining how this technology helps secure software execution against curious entities and powerful attackers.

Part I

Background

Chapter 2

Trusted execution environments

Trusted execution environments (TEEs) are a cornerstone security technology of this dissertation, providing hardware-based primitives that protect the confidentiality of sensitive data and computations across cloud and IoT infrastructures. This chapter presents a comprehensive survey of the many TEE paradigms and technologies developed over the past decade, including Intel SGX, Arm TrustZone, AMD SEV, and some RISC-V implementations. Finally, the chapter also mentions some limitations and drawbacks, which hinder the widespread adoption of TEEs.

Chapter outline

2.1 A hardware-assisted security system with many faces 20
2.2 A structured comparison of TEE characteristics and features 22
2.2.1 Introduction to Intel SGX 23
2.2.2 Introduction to Arm TrustZone 27
2.2.3 Introduction to AMD SEV 30
2.2.4 Introduction to other notable TEE implementations 34
2.2.5 A summary of the TEE characteristics 36
2.3 Are TEEs the silver bullet for security? 36

2.1 A hardware-assisted security system with many faces

Security is a paramount concern in today's interconnected digital landscape, particularly in the era of cloud, edge, and IoT systems, where users' data is distributed across multiple locations under different jurisdictions. Cloud users must trust that their data and workloads remain secure on shared infrastructure, despite potential vulnerabilities and curious providers with administrative power. Simultaneously, infrastructure providers need protection from malicious tenants who may exploit vulnerabilities for personal gain. The distributed nature of edge computing further complicates security, as devices are installed in various locations, making physical control challenging. Edge administrators have similar powers to cloud providers, while users in proximity may physically abuse devices. Addressing these security challenges is crucial for ensuring the integrity and confidentiality of data and systems in modern computing environments.

To address the security challenges in cloud and edge computing, various privacy-preserving technologies have been proposed. Fully homomorphic encryption (FHE) allows computation on encrypted data without the need for decryption, but it incurs significant computational overheads that make it impractical for many real-world applications [55–57]. Secure multi-party computation (SMPC) enables multiple parties to collaborate on a computational task while keeping their individual inputs private, but the extensive communication required between the parties results in substantial network overheads, limiting its scalability and performance [58–60]. Trusted computing (TC) with Trusted Platform Modules (TPMs) offers a hardware-based solution for secure key storage and attestation, but it lacks support for arbitrary mathematical operations and code execution, restricting its applicability in privacy-preserving computations [61]. While these technologies provide strong security guarantees, their performance limitations and lack of versatility have motivated the search for alternative approaches that can better balance security and efficiency in privacy-preserving computations.

Most recent versions of popular computer architectures include some form of a trusted execution environment (TEE), a practical solution for *isolating* software execution and its related data. TEEs aim to provide safe and trustworthy code execution on (remote) untrusted hardware. Hardware manufacturers have provided TEE implementations more than a decade ago, each one of them offering different features and guarantees. These technologies enable the processing of data, contained in isolated memory areas that cannot be accessed or tampered with by more privileged software, such as the operating system or the hypervisor. Hence, cloud providers and edge device owners with management rights or even physical control cannot access the data and computation of a tenant, protecting the confidentiality of their applications.

Varying trust boundaries The absence of a unified definition for TEEs is a consequence of the lack of standardisation across TEE implementations. Instead, each TEE is tied to a specific threat model, as defined by the respective CPU vendor. However, CPU manufacturers have introduced a variety of TEE designs over the years, which have converged towards a common set of trust boundaries and security features across different TEE implementations. Figure 2.1 illustrates the legacy computing stack in contrast to the three primary trust boundaries found in current TEE implementations when handling confidential data. The green box represents the trusted

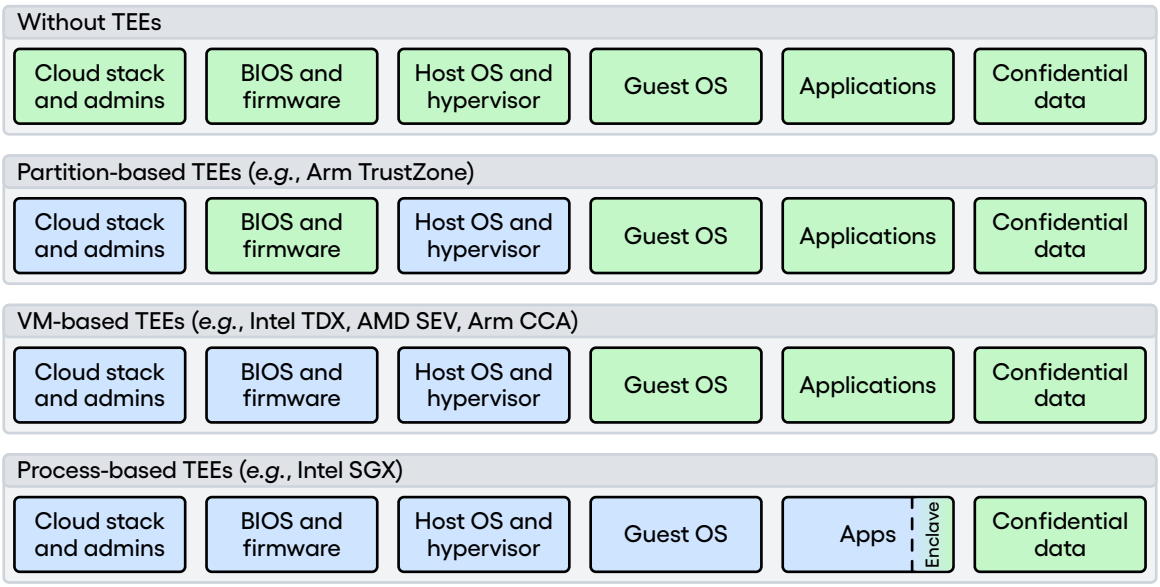


Figure 2.1: The trust boundaries of the legacy computing stack (without TEEs) compared to the three primary trust boundaries in current CPU-bound TEE implementations. The green boxes (▢) denote the trust boundaries, *i.e.*, the elements with potential access to confidential data, whereas the blue boxes (▢) indicate potentially malicious components that do not compromise data confidentiality. Fewer green boxes are better from a security standpoint, as it reduces the attack surface of the TCB.

computing base (TCB), which is the portion of the environment that must be trusted, as it has access to confidential data. These environments are elaborated upon in the following sections.

- **Legacy stack.** In the absence of confidential computing, the entire environment must be trusted, including the premises, staff, hardware, operating systems (OSes), and applications handling confidential data. Data owners are required to trust the machine operators and the software stack, and a vulnerability at any level can lead to data breaches.
- **Partition-based TEEs.** This approach divides the system into two or more partitions, relying on the processor to ensure secure isolation. The primary benefit of this trust boundary is the exclusion of machine operators and the regular operating system from the TCB. However, this approach typically requires the use of a small, dedicated, and trusted operating system within the trusted partition. A significant drawback is the resulting large TCB, which comprises various software such as the firmware, operating system, and unrelated applications in the same environment as the confidential data. The increased TCB size increases the risk of security vulnerabilities in the code [110]. Examples of partition-based TEEs include Arm TrustZone [104] and Keystone [105] for RISC-V.
- **VM-based TEEs.** Virtual machines (VMs) have proven to be an effective paradigm for isolating operating systems and their applications, preventing them from altering the hosting environment. VM-based TEEs extend this virtualisation concept by transforming trusted VMs into a two-way sandbox, preventing the hypervisor and host OS from interacting with the trusted VM. This trust boundary is the most natural for operators and develop-

ers, as most operating systems and applications are already compatible with this approach. Similar to partition-based TEEs, VM-based TEEs suffer from a large TCB due to the embedding of the operating system and unrelated software. Examples of VM-based TEEs include AMD SEV [106], Intel TDX [107], and Arm CCA [108, 153].

- **Process-based TEEs.** Process-based TEEs limit the trust boundary to only the code that processes the confidential data, resulting in the smallest TCB. This reduced attack surface provides a security advantage, as it reduces the likelihood of facing vulnerable software components. However, this approach often impacts the ease of programming trusted applications, requiring a shift in the development paradigm since the operating system is not trusted. This leads to severe constraints, such as the inability to trust system calls. Examples of process-based TEEs include Intel SGX [102] and Sanctum [103] for RISC-V.

The remainder of this chapter explores three widely-used market-available CPU-bound TEE implementations: Intel SGX [102], Arm TrustZone [104], and AMD SEV [106]. Additionally, this chapter briefly documents several other academic and emerging TEE implementations to provide a comprehensive overview. Intel SGX and Arm TrustZone are further used in this dissertation as the foundation for implementing the trusted WebAssembly (Wasm) runtimes Twine and WaTZ, which are discussed in detail in Chapters 6 and 7, respectively.

2.2 A structured comparison of TEE characteristics and features

In this section, we present a structured comparison of the cornerstone features of TEEs. The following characteristics are compared for each TEE implementation.

- **Memory protection**: Memory protection is a core feature of any TEE. Its primary purpose is to safeguard the confidentiality of the memory used by trusted software components. TEEs use several mechanisms to achieve this goal. *1) Memory isolation*: segregates the trusted memory region from untrusted software, ensuring that sensitive data remains inaccessible to unauthorised processes. *2) Memory integrity*: an active mechanism that prevents the TEE memory from being tampered with. *3) Memory encryption*: protects against unauthorised access and memory snooping attempts, adding an extra layer of security in case the memory isolation is breached or a hardware attack is conducted. *4) Memory freshness*: protects against replay and rollback attacks, ensuring that the integrity of the TEE state is preserved throughout its lifecycle.
- **Development paradigm**: The development paradigm for trusted applications is a key aspect to consider when opting for a TEE implementation. It comprises the programming approaches and the complexity involved in creating software that runs within the TEE.
- **Execution flow**: Describe the process of starting the TEE instance, loading the trusted application, and managing its execution and interaction with the untrusted environment.
- **Concurrent domains**: TEE instances may run concurrently on the same machine while the hardware guarantees the trust boundaries between these instances.

- **System support for isolation**: The hardware and software mechanisms that enforce the TEE features, protecting the trusted software from the untrusted environment.
- **Open source**: Indicate whether the source code of the TEE components is either partially or fully publicly available, allowing scrutiny from the community.

2.2.1 Introduction to Intel SGX

Intel Software Guard Extensions (SGX) [102] is a security extension of the Intel instruction set architecture (ISA) introduced in 2015 with the Skylake generation of Intel processors. SGX enables the creation of encrypted memory regions called *enclaves*, which are executed in user mode and mapped into the virtual address space of the host process. The purpose of enclaves is to provide a secure environment for executing code and processing sensitive data, shielded from the outside world, including the operating system, the hypervisor and privileged users.

Intel SGX uses a dedicated memory region called the Enclave Page Cache (EPC) to store the code and data of enclaves. The EPC is a secure area within the system memory that is inaccessible to other programs, including the operating system and hypervisor. When an enclave is created or accessed, its pages are loaded into the EPC, and the traffic between the CPU and the EPC remains confidential thanks to the memory encryption engine (MEE). However, the EPC has a limited size, which varies depending on the SGX version. When the EPC becomes full, and additional enclave pages need to be loaded, the processor uses a swapping mechanism to evict some of the existing pages from the EPC and replace them with the required ones, leading to performance degradation due to the overhead associated with the swapping process [154, 155].

Intel SGX has evolved through multiple versions in two dimensions: instruction set and implementation [156]. The former axis comprises *SGX1* and *SGX2*, while the latter axis includes *SGX Client* and *SGX Scalable*. The main differences between SGX1 and SGX2 are explained below.

- **SGX1**: Introduced in the Skylake generation of Intel processors in 2015 and available up to the Kaby Lake generation in 2016, SGX1 represents the initial implementation of Intel SGX, providing the core features for creating and executing secure enclaves with a fixed amount of enclave memory set during the compilation of the enclave binaries.
- **SGX2**: Introduced in the Gemini Lake generation of consumer-grade processors in 2017 and the Ice Lake generation of server-grade processors in 2019, SGX2 [157, 158] allow additional flexibility in runtime management of enclave resources, *e.g.*, Enclave Dynamic Memory Management (EDMM), and thread management within an enclave [159].

The main differences between SGX Client and SGX Scalable are explained below.

- **SGX Client**: Initially shipped with SGX1, SGX Client has an EPC limited to the small size of 256 MiB, leading to performance degradation as the EPC usually reaches capacity using any real-world application, triggering the swapping mechanism of enclave memory pages.

- **SGX Scalable**: Introduced in the Ice Lake generation of server-grade processors in 2019, SGX Scalable [160, 161] supports multi-socket platforms and extends the EPC up to 512 GiB per CPU, significantly reducing the need for swapping for large enclaves. However, this increased capacity comes with drawbacks that are elaborated below.

Memory protection Intel SGX isolates the memory of enclaves from the untrusted environment by leveraging hardware-based access control mechanisms. The processor enforces strict access control policies to ensure that only authorised code executing within an enclave can access the enclave's memory. When an enclave is created, the processor allocates a dedicated portion of the EPC to store the enclave's code and data. The processor maintains a set of access control structures, such as the Enclave Page Cache Map (EPCM), which keeps track of the permissions and ownership of each EPC page. The processor checks these access control structures on every memory access to determine whether the access is allowed. If the code executing outside an enclave attempts to access the enclave's memory directly, the processor will deny the access, effectively isolating the enclave's memory from the untrusted environment.

SGX Client and SGX Scalable handle memory encryption, integrity, and freshness differently.

- In **SGX Client**, the MEE transparently encrypts enclave memory data blocks (512 bit, the size of a cache line) using AES-CTR, a symmetric block cipher, when they leave the CPU cache lines [162]. The encrypted data blocks are stored in the EPC and only decrypted when loaded back into the CPU, ensuring data confidentiality against software and hardware attacks. Moreover, SGX Client authenticates each data block with a message authentication code (MAC) and stores them in the EPC, which guarantees data integrity against tampering attacks. To prevent rollback attacks, SGX Client maintains an integrity tree (a variant of a Merkle tree), containing counters that are associated with the encryption of data blocks. While the lower levels of the integrity tree (*i.e.*, those closest to the leaf nodes) reside in the EPC, the upper levels (a few kilobytes) are stored in the processor package's SRAM, which is resilient to software and hardware rollback attacks [162]. When the EPC reaches its capacity, the OS evicts some enclave memory pages to untrusted memory, using similar cryptographic protections to ensure the confidentiality, integrity, and freshness of the evicted pages [163].
- **SGX Scalable** uses Intel Total Memory Encryption (TME) [164] technology to encrypt the memory of enclaves. Intel TME is a hardware-based security feature that encrypts the entirety of a system's physical memory using ephemeral keys generated by the processor. When used with SGX Scalable, TME uses a separate key for encrypting the EPC. TME relies on the AES-XTS encryption mode, which provides confidentiality but does not inherently provide integrity and freshness against hardware attacks, as this block cipher does not authenticate ciphertext. By removing the additional memory access required for the integrity tree, SGX Scalable reduces the performance impact to a small latency on each memory operation due to encryption. As a result, SGX Scalable trades some hardware guarantees with support for larger EPC sizes while balancing the performance requirements of scalable server workloads [160].

Development paradigm An SGX enclave program is partitioned into two distinct components: the untrusted part and the trusted part. The untrusted part, built as a regular executable, contains all code that does not operate on sensitive data at runtime. Conversely, the trusted part, compiled as a shared object, contains the enclave code, which operates on sensitive data within the TEE, *i.e.*, the isolated region mapped in the application's virtual address space. The enclave binary is self-contained and includes all the code and dependencies required for secure execution, operating independently and without relying on any operating system services, such as system calls, as the OS resides outside the trust boundary of a partition-based TEE.

Building Intel SGX enclave programs requires using a unique programming paradigm that differs from traditional application development. Developers can leverage various tools and frameworks to create these programs, each with its advantages and trade-offs. The official Intel SGX SDK [165] provides a rich set of libraries, tools, and documentation for developing enclave applications in C/C++. Alternatively, developers can use unofficial SDKs that abstract the target TEE, such as Open Enclave [133], or leverage SDKs tailored for other programming languages, like the Apache Teaclave SGX SDK for Rust [166], which provides a more expressive programming model. Another approach is to use library OSes like SGX-LKL [64], Gramine [63] or Occlum [167], which abstract the complexity of the SGX development paradigm by wrapping existing applications and mimicking the system interface of a traditional OS. Regardless of the chosen approach, developing Intel SGX enclave programs remains a complex task that requires careful consideration of the security implications of specific implementations.

Execution flow Figure 2.2 shows the execution flow of an SGX program. The untrusted part, built as a standard executable, creates the enclave by allocating and configuring the necessary control structures in the EPC. Code and data pages are then added to the enclave (①). Once initialised, the host process can invoke the enclave's trusted functions through enclave calls (ECALLs), which transition the logical processor into enclave mode and securely transfers the execution flow to predefined entry points within the trusted code (②). Conversely, the enclave can use outside calls (OCALLs) to execute functions in the untrusted part when necessary (③). Upon completion of a trusted function, the execution flow returns to the caller in the untrusted part (④). Hardware exceptions during enclave execution are handled by a mechanism that saves the enclave state and invokes the exception handler before resuming execution.

Concurrent domains Intel SGX does not impose a strict limit on the number of enclaves that can be concurrently active on a single system. However, the capacity of the EPC does constrain the total amount of protected memory available for all running enclaves. To mitigate this limitation, the OS can evict enclave memory pages from the EPC and store them in unprotected main memory, thereby freeing up EPC memory for newly created enclaves. The drawback of EPC page eviction is increased latency when accessing swapped-out enclave pages. Before an evicted page can be accessed by the enclave, it must first be moved back into the EPC, incurring a performance penalty, estimated to 40 k cycles, compared to 200 cycles on EPC hit [155]. With SGX Scalable, which supports a considerably larger EPC, the need for enclave page eviction is greatly reduced, improving performance for workloads requiring many large enclaves.

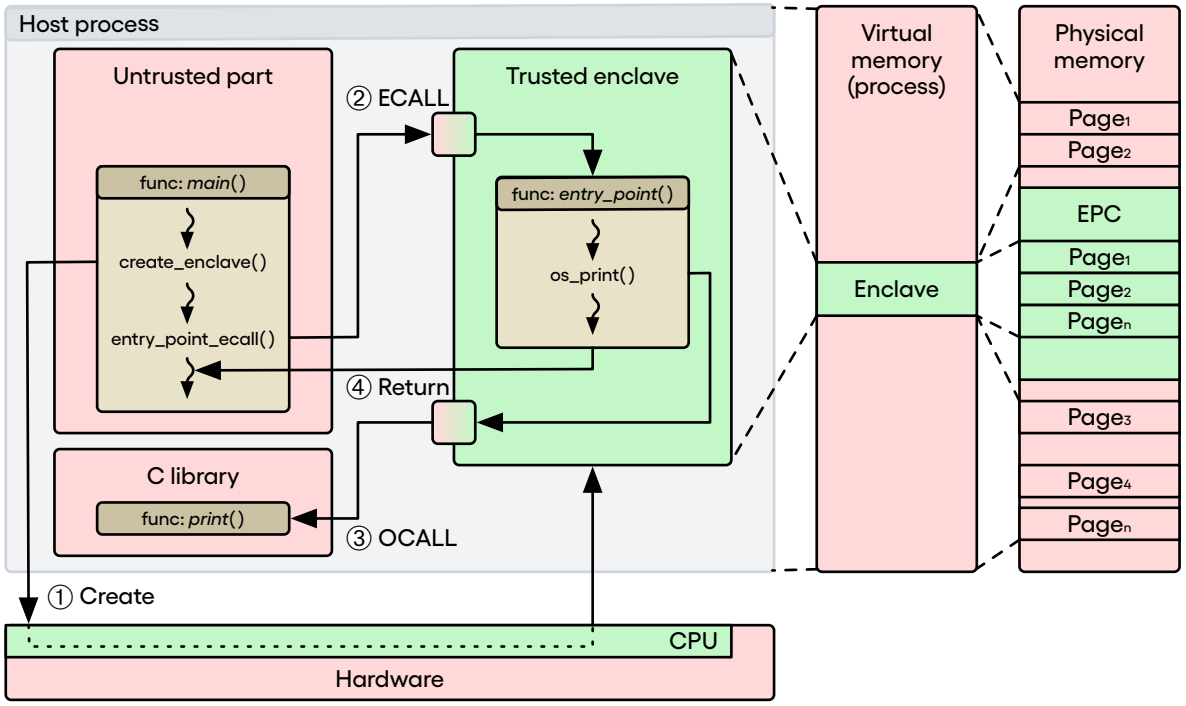


Figure 2.2: The typical execution flow of running a trusted application using Intel SGX. The green boxes (▢) illustrate the TCB of Intel SGX, while the red boxes (▢) represent components outside the TCB, which can be potentially compromised or malicious.

System support for isolation The protection is enforced through a combination of hardware and software components. At the hardware level, SGX introduces new instructions in Intel ISA that form the foundation of the SGX implementation. These instructions are implemented using XuCode [168], a technology developed by Intel to deliver complex instruction flows more quickly and with reduced impact compared to traditional hardware implementations. XuCode runs from protected system memory in a special execution mode of the CPU, set up by processor firmware. It has its own instruction set based on the 64-bit instruction set. Intel SGX includes the SGX instruction leaf flows, memory management, enclave operations, key derivation, and TCB recovery, which are implemented using a combination of XuCode, microcode, and hardware circuits. Finally, these low-level mechanisms are invoked or managed by software, using an SGX kernel driver and the SGX Platform Software (PSW), with the latter consisting of user-level services and libraries that enable interaction between SGX enclaves and the host system.

Open source Intel SGX's hardware components, including the XuCode and microcode, are proprietary and not disclosed to third parties. However, recent reverse engineering efforts have revealed some of these microarchitectural implementation details [169, 170]. In contrast to the closed-source hardware, most SGX software implementations are open-source. These include the SGX driver, which has been integrated into the Linux kernel, as well as the SGX PSW, the SGX SDK, and various tools for issuing and verifying SGX attestation [171].

2.2.2 Introduction to Arm TrustZone

TrustZone [104] is a hardware security extension introduced by Arm in 2004 for application processors (Cortex-A series), known as *TrustZone-A*, and later adapted for microcontrollers (Cortex-M series), referred to as *TrustZone-M*. It provides a secure execution environment by partitioning the processor into two worlds: the normal world and the secure world. The normal world runs a rich operating system and untrusted applications, while the secure world hosts security-critical services and trusted applications. The isolation between the two worlds is enforced by hardware, ensuring that the normal world cannot access the resources of the secure world.

While TrustZone-A and TrustZone-M share the same high-level concepts, they differ in their low-level implementations to accommodate the specific requirements of their target platforms.

- **TrustZone-A** (TrustZone for Cortex-A processors) [172], starting with the Armv6K architecture, provides a single secure execution environment per system, managed by a secure monitor that handles the context switching between the normal and secure worlds. It allows the secure world to access resources of the normal world, while the normal world is prevented from accessing secure resources directly.
- **TrustZone-M** (TrustZone for Cortex-M processors) [173], starting with the Armv8-M architecture, is designed for resource-constrained embedded devices and features a more lightweight implementation. It removes the need for a secure monitor by using memory map-based partitioning and automatic state transitions in exception handling code.

OP-TEE (Open Portable Trusted Execution Environment) [111] is an open-source framework that leverages TrustZone-A to provide a standard and secure environment for running trusted applications (TAs). It is one of the several ways to create and execute TAs for TrustZone, and this thesis focuses on the OP-TEE approach. OP-TEE consists of a small trusted OS running in the secure world and an API for communication between the normal and secure worlds. The trusted OS manages the lifecycle of TAs and provides services such as secure storage, and cryptographic operations. OP-TEE follows the GlobalPlatform TEE specifications, which define a standardised architecture and API for developing TAs that can be ported across different TEEs platforms.

Memory protection TrustZone provides hardware-enforced memory isolation between the secure world (TEE) and normal world (REE), with the isolation extending from the processor core to the memory and peripheral buses. The isolation is achieved through the use of a dedicated processor bit, known as the non-secure (NS) bit, and determines the current security state of the processor. The memory infrastructure is extended with the TrustZone Address Space Controller (TZASC) and TrustZone Memory Adapter (TZMA), which use the NS bit to configure specific memory regions as secure or non-secure. When the NS bit is set, indicating that the processor is in the normal world, any attempt to access secure memory regions is blocked. Conversely, when the NS bit is cleared, the processor operates in the secure world and can access both secure and non-secure memory. This strict memory partitioning ensures that even if the REE is compromised, it cannot access sensitive data residing in secure memory.

However, while TrustZone offers robust memory isolation, it does not inherently guarantee the confidentiality, integrity, or freshness of data stored in secure memory. The absence of memory

encryption leaves data potentially vulnerable to physical attacks by adversaries with direct access to the memory bus or DRAM modules. Moreover, TrustZone lacks native support for memory integrity verification and replay protection, which are crucial for preventing unauthorised tampering. To achieve these security properties, additional system-on-chip (SoC) extensions are typically required. These extensions may include dedicated hardware modules for memory encryption and secure boot mechanisms to verify the integrity of the TEE.

Development paradigm Developing TAs for TrustZone involves a specific programming approach and introduces additional complexity compared to regular application development. TAs are typically written in C and compiled into a single binary that runs within the secure world managed by a trusted OS like OP-TEE. Alternative development frameworks exist, such as RusTEE [67] for programming TAs using Rust. To be trusted by OP-TEE, the binary must be signed with a private key corresponding to a public key that OP-TEE recognises. The signed binary can either be statically compiled into OP-TEE or located in the REE file system, from where OP-TEE will load it into secure memory when needed. TAs have direct low-level access to secure world resources but are constrained by the limited memory and API provided by the trusted OS. This restriction prevents TAs from leveraging the libraries and system calls available in the normal world OS. They can communicate with client applications in the normal world through a secure monitor that mediates world switches and data sharing. Debugging a trusted application requires either an emulator like QEMU or a special-purpose development board equipped with a TrustZone-enabled processor. These additional steps and requirements make the development and testing cycle for TAs more complex compared to traditional applications.

Execution flow Figure 2.3 illustrates the execution flow of a TA in TrustZone using OP-TEE. A TA is initialised when a program running in the normal world invokes the TA using the GlobalPlatform (GP) API (①). This API call is handled by the OP-TEE driver in the Linux kernel, which forwards the request to the OP-TEE OS running in the secure world via a Secure Monitor Call (SMC) (②). The OP-TEE OS resolves the TA using its universally unique identifier (UUID), which is passed as a parameter in the GP API call. The secure monitor is then used to switch the processor state from the normal world to the secure world (③), where the OP-TEE OS takes control (④). The OP-TEE OS then locates the TA binary, verifies its signature, and loads it into the user space memory. Once the TA is initialised, the OP-TEE OS invokes a predefined entry point within the trusted code, allowing the TA to execute in the secure user space (⑤). During its execution, the TA can request services from the normal world OS using remote procedure calls (RPCs) handled by the *tee-supplicant* daemon, running in the normal world user space (⑥).

Concurrent domains The TrustZone architecture is limited to running a single secure world concurrently alongside the normal world. The processor can only be in one world at a time, as indicated by the NS bit, which is propagated throughout the system to enforce strict isolation between the two worlds. Consequently, TrustZone does not natively support the execution of multiple secure worlds in parallel, as the secure monitor is responsible for managing the context switching between the two worlds based on the NS bit. This architectural constraint has led researchers to explore virtualisation techniques to extend TrustZone's features and enable the co-existence of multiple secure environments on the same device. Notable examples include

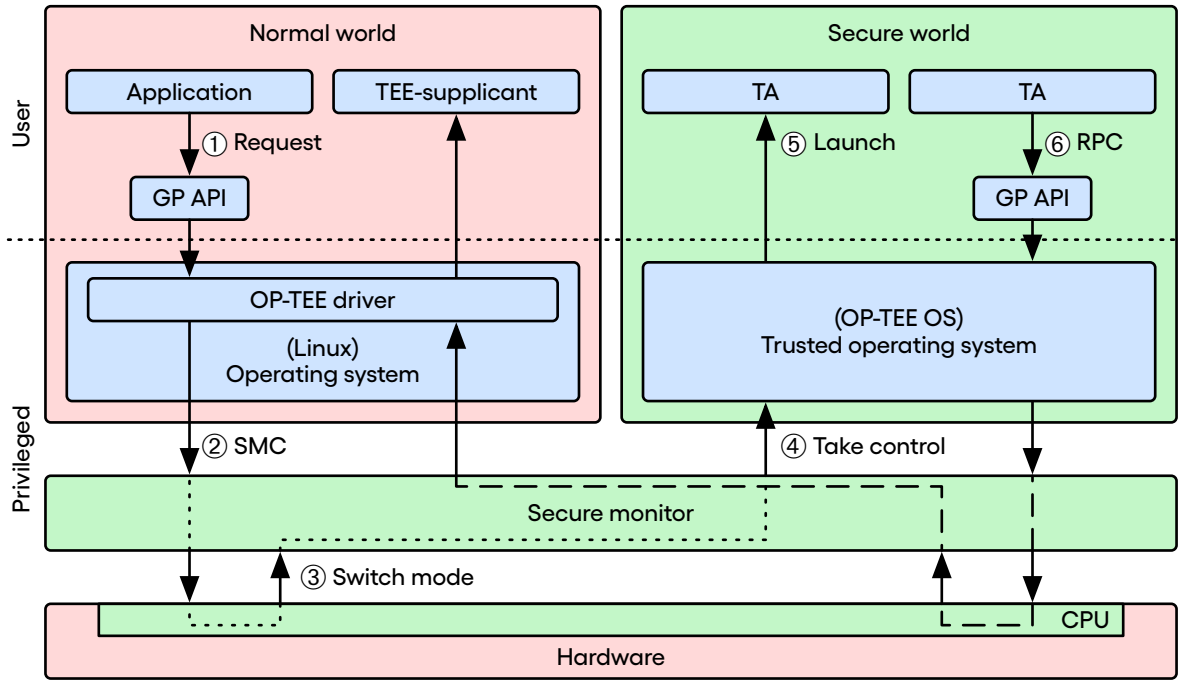


Figure 2.3: The typical execution flow of running a trusted application using Arm TrustZone with OP-TEE. The green boxes (▢) illustrate the TCB of Arm TrustZone, while the red boxes (▢) represent components outside the TCB, which can be potentially compromised or malicious.

RTZVisor [174] and µRTZVisor [175], which leverage TrustZone to create a virtualised environment in the secure world. These hypervisors allow multiple guest operating systems to run concurrently within the secure world, each with its own isolated memory space.

System support for isolation TrustZone's isolation between the secure and normal worlds is enforced through a combination of processor, memory, and peripheral protection mechanisms. The processor operates in either the secure or normal state, as determined by the NS bit. This bit is propagated throughout the system buses to enforce access control. During the boot process, the firmware configures the TrustZone TZASC and TZMA to partition memory into secure and non-secure regions. The secure monitor, running in the highest privilege level, is responsible for managing world switches via the SMC instruction. The monitor saves the state of the current world and restores the state of the target world during these transitions. The NS bit is used by the memory controllers to prevent non-secure access to secure memory regions. Peripherals can also be configured as secure or non-secure using the TrustZone Protection Controller (TZPC). This allows the secure world to have exclusive access to a subset of peripherals.

Open source Open-source reference implementations of the firmware for Arm SoCs are widely accessible to developers and researchers, such as Trusted Firmware-A (TF-A) [176] for Cortex-A processors and Trusted Firmware-M (TF-M) [177] for Cortex-M processors. These projects provide a solid starting point to build upon, and many public toolchains for building trusted systems rely on them. Additionally, there are open-source initiatives for trusted operating sys-

tems, such as OP-TEE [111] and Open-TEE [178], which implement the necessary framework for programming and securing TAs and enabling communication between the normal and secure worlds. In contrast to the software stack, the openness of TrustZone hardware is largely dependent on the manufacturers [179]. Some manufacturers, such as Xilinx, have taken a more transparent approach by publicly disclosing technical details and releasing development boards [180]. However, other manufacturers have been more reluctant to openly share technical details, making their platforms less suitable for academic exploration of TrustZone [181].

2.2.3 Introduction to AMD SEV

AMD Secure Encrypted Virtualization (SEV) is a hardware-based technology that enables the creation of confidential VMs on AMD processors. SEV allows the execution of minimally modified operating systems and unmodified applications within an isolated environment, shielding the VM's memory from the hypervisor and other privileged software. The primary purpose of SEV is to protect the confidentiality of data processed within the VM, even in untrusted cloud environments. SEV is available in three versions: SEV, SEV-ES, and SEV-SNP, each building upon the previous version to enhance security and address specific vulnerabilities.

- **SEV** [182]: Introduced with the 1st-generation AMD EPYC processor family (2017), SEV provides memory encryption for VMs using a unique 128-bit AES key per VM, ensuring the confidentiality of data processed within the VM. However, this version of SEV could leak sensitive information during interrupts from guests to the hypervisor through unencrypted registers, exposing the VM's internal state to a malicious or compromised hypervisor [183].
- **SEV-ES** [184]: Introduced with the 2nd-generation AMD EPYC processor family (2019), SEV-ES (Encrypted State) addresses the register leakage vulnerability present in the initial version of SEV, where sensitive information could be exposed during interrupts from guests to the hypervisor through unencrypted registers. SEV-ES encrypts the VM's register state during transitions between the guest and the hypervisor, ensuring that the hypervisor can only access specific guest registers with explicit permission from the guest OS.
- **SEV-SNP** [185]: Introduced with the 3rd-generation AMD EPYC processor family (2021), SEV-SNP (Secure Nested Paging) builds upon SEV-ES by adding strong memory integrity protection to prevent hypervisor-based attacks [123, 186, 187]. SEV-SNP mitigates threats such as memory tampering and TCB rollback, addressing the limitations of SEV-ES, which was vulnerable to certain attacks that could compromise the trust that third parties could place in confidential VMs when operating in untrusted environments remotely.

Memory protection The memory protection mechanisms of AMD SEV, SEV-ES, and SEV-SNP are incrementally built upon each other, with each version introducing additional security features and enhancements that are further elaborated below.

- **SEV**: Relies on the underlying technology of AMD Secure Memory Encryption (SME) to isolate the memory of confidential VMs. SME is a hardware-based feature that enables the encryption of main memory to protect against physical attacks and unautho-

rised access. It uses dedicated hardware in the on-die memory controllers, which include a high-performance Advanced Encryption Standard (AES) engine. This engine encrypts data when it is written to DRAM and decrypts it when read, using a unique 128-bit AES key managed by the AMD Secure Processor (AMD-SP). The AMD-SP is a dedicated co-processor integrated within the AMD SoC, responsible for generating and storing the encryption key securely. The OS or hypervisor controls which memory pages are encrypted by setting the C-bit (a dedicated bit of the physical address) in the page table entries, indicating that the corresponding pages should be encrypted. AMD SEV extends SME by assigning a unique encryption key to each VM, allowing the guest OS to choose which pages to encrypt selectively. This enables the encryption of sensitive data while leaving some pages unencrypted for data exchange with other VMs or peripherals. Additionally, SEV introduces the concept of VM address space identifier (ASID), a unique identifier assigned to each VM by the hypervisor, which is used to tag all code and data associated with a particular VM, ensuring that the data can only be accessed by the VM owner. As a result of this fine-grained security model, both the hypervisor and guest OSes must be designed to support and leverage these hardware-based security features.

- **SEV-ES**: The first version of AMD SEV has a weakness where sensitive information can be exposed during interrupts from guests to the hypervisor through unencrypted registers. For example, a malicious hypervisor can read the registers typically used to hold AES keys when using the x86 AES instructions, potentially leading to data leaks. SEV-ES addresses this issue by encrypting the guest VM's register state during transitions between the guest and the hypervisor. It introduces the concept of a Virtual Machine Control Block (VMCB), which is divided into two sections: the *control area* owned by the hypervisor and the *save area* used to store the VM's encrypted register state. The guest VM controls which registers are exposed to the hypervisor on a per-case basis, ensuring that the hypervisor can only access specific guest registers with explicit permission from the guest OS.
- **SEV-SNP**: While SEV-ES offers additional protection for CPU register state, it still has weaknesses that can be exploited by a malicious hypervisor. For instance, the hypervisor can perform memory rollback attacks, where it replaces the current memory content with an older snapshot, leading to data corruption or the execution of outdated code. Additionally, attackers may attempt to roll back the AMD-SP firmware itself to a vulnerable version, compromising the security of the entire system. SEV-SNP tackles these issues by introducing memory integrity protection through the Reverse Map Table (RMP), which tracks the ownership of each memory page and ensures that only the owner can modify its content. The RMP, combined with a new page validation mechanism, prevents memory replay, corruption, aliasing, and re-mapping attacks. Furthermore, SEV-SNP enhances the security of the AMD-SP firmware by cryptographically signing its version, leading to a new proof of trust called the Versioned Chip Endorsement Key (VCEK), making it impossible for attackers to rollback to an older, vulnerable version without notice. Another notable improvement in SEV-SNP is that guest operating systems can now directly communicate with the AMD-SP using a protected path, enabling advanced features such as flexible attestation and secure key derivation. It is important to note that while SEV-SNP significantly enhances the security of confidential VMs, it does not address physical on-

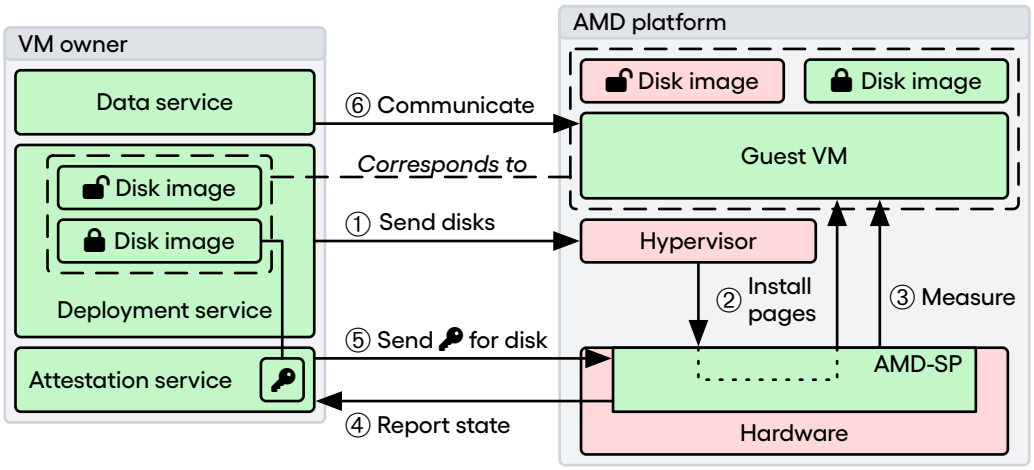


Figure 2.4: The typical execution flow of running a virtual machine using AMD SEV. In the right part *AMD platform*, the green boxes (▢) illustrate the TCB of AMD SEV, while the red boxes (▢) represent components outside the TCB, which can be potentially compromised or malicious.

line attacks on the memory subsystem, which are considered out of scope due to their complexity, matching the same physical security guarantees as SGX Scalable.

Development paradigm One of the key advantages of VM-based TEEs like AMD SEV is that the programming paradigm for trusted applications remains largely unchanged compared to traditional, unencrypted VMs. Developers can continue to use familiar tools, frameworks, and programming languages to build their applications. However, it is important to note that applications running within a confidential VM should still be aware of their secure execution environment. This awareness is necessary for the application to prove its trustworthiness to third parties, such as when receiving confidential data for processing. To facilitate this, the guest OS plays a crucial role in handling TEE interactions with the hypervisor, abstracting low-level details from the application layer. Well-established operating systems like Linux and Windows already provide support for AMD SEV, further easing the development process.

Execution flow Figure 2.4 depicts the execution flow of an AMD SEV VM with a trusted application. The process begins with the VM owner providing two disk images: an unencrypted one containing the guest operating system and an encrypted one with the software and data (①). The hypervisor requests the AMD-SP to install this initial set of pages in the guest VM (②). The AMD-SP then cryptographically measures the initial state and layout of the guest memory, generating an attestation report containing signed information associated with the VM (③). The attestation report is then communicated back to the VM owner for verification (④). If the attestation report is deemed valid, the VM owner provides the necessary confidential data and keys, which may include a decryption key for the second encrypted disk image containing sensitive components of the VM, like the trusted application (⑤). Once the VM is fully initialised, the trusted application is launched within the guest OS, following the same process as a traditional application. The trusted application can then securely communicate with the VM owner to fetch additional confidential data or keys as needed (⑥). Aside from the secure initialisation, attes-

tation, and communication with the VM owner, the execution flow of the trusted application within the AMD SEV environment remains largely unchanged compared to non-SEV setups.

AMD SEV-SNP introduces some improvements over previous versions by supporting flexible attestation. In contrast to SEV and SEV-ES, which only allow attestation during the initial guest launch, SEV-SNP enables the guest VM to request attestation reports from the AMD-SP at any time during its execution. This is made possible by a protected communication path between the guest VM and the AMD-SP. The guest VM can use this secure channel to request attestation reports and cryptographic keys. Attestation is further covered in Chapter 3.

Concurrent domains The number of concurrent confidential VMs that can run on a system using SEV is determined by the processor generation, as the constraints have evolved with each new release of AMD EPYC processors [188]. The parameters *SEV-ES ASID Count* and *SEV-ES ASID Space Limit* dictate the maximum number of concurrent VM instances. SEV-ES ASID Count indicates the total number of ASIDs allocated for both SEV and SEV-ES VMs, while SEV-ES ASID Space Limit determines the number of these ASIDs reserved specifically for SEV-ES VMs. These settings are automatically adjusted and can be manually configured in the BIOS based on the amount of system memory. When the system has 8 TiB or more of DRAM, the BIOS reduces the ASID count to 253 to accommodate the increased memory capacity, because the ASIDs require dedicating a part of the physical address bits to uniquely identify each VM, limiting the available address space for DRAM. The limitations per processor family are further elaborated as follows:

- **1st-gen EPYC processors** only support SEV with 15 ASID keys.
- **2nd-gen EPYC processors** support both SEV and SEV-ES with either 509 ASID keys for systems with up to 8 TiB DRAM or 253 ASID keys for systems with up to 16 TiB DRAM.
- **3rd-gen EPYC processors** support SEV, SEV-ES, and SEV-SNP with either 509 ASID keys for systems with up to 8 TiB DRAM or 253 ASID keys for systems with up to 16 TiB DRAM.
- **4th-gen EPYC processors** support SEV, SEV-ES, and SEV-SNP with 1006 ASID keys.

System support for isolation The *memory protection* paragraph covers many aspects of the underlying security systems that AMD SEV relies on. In addition to the mechanisms previously described, the input/output memory management unit (IOMMU) plays a key role in enforcing the memory access of the confidential VMs [189]. The IOMMU is a hardware component that manages memory access for devices, and it enforces the memory access control policy to block unauthorised access to encrypted VM pages. This ensures that DMA requests respect the ownership and permissions of memory pages, protecting against malicious devices attempting to access confidential VM memory. The AMD-SP is responsible for managing the RMP and communicating with the IOMMU to keep it synchronised with the current memory permissions state. Furthermore, the AMD-SP firmware can be upgraded upon discovery of vulnerabilities, and starting with SEV-SNP, its version is included in attestation reports, enabling guest owners to verify that their VMs are running on a platform with an up-to-date firmware version.

Open source The software stack supporting AMD SEV is largely open source. The Linux kernel, for example, includes an open-source SEV module that acts as a mediation layer between the guest operating system and the hypervisor [190]. Hypervisors like KVM and QEMU have open-source implementations of SEV support, allowing users to deploy confidential VMs in their environments [191, 192]. Furthermore, AMD provides open-source tools and libraries, which facilitate interactions with SEV-enabled guest VMs, including managing the launch process, performing attestation, and obtaining cryptographic materials [193, 194].

Unlike the open-source software stack, the hardware implementation of AMD SEV, including the AMD-SP firmware and other low-level hardware details, is proprietary and not publicly disclosed. Despite this, AMD provides detailed documentation on the SEV architecture, allowing researchers and developers to understand the system threat model and security properties.

2.2.4 Introduction to other notable TEE implementations

Keystone An academic and modular RISC-V framework that provides the building blocks to create TEEs, rather than providing an all-in-one solution that is inflexible and is another fixed design point [105]. Instead, the authors advocate that hardware should provide security primitives instead of point-wise solutions. Keystone implements a secure monitor at machine mode (M-mode) and relies on the RISC-V Physical Memory Protection (PMP) [195] instructions to provide isolated execution and, therefore, does not require any hardware change. PMP is a part of the RISC-V Privileged Architecture specifications and describes the interface for a standard RISC-V memory protection unit. Since Keystone leverages feature composition, the framework users can select their own set of security primitives, *e.g.*, memory encryption, dynamic memory management and cache partitioning. Each trusted application executes in user mode (U-mode) and embeds a runtime that executes in supervisor mode (S-mode). The runtime decouples the infrastructure aspect of the TEE (*e.g.*, memory management, scheduling) from the security aspect handled by the secure monitor. As such, Keystone programmers can roll their custom runtime to fine-grained control of the computer resources without managing the TEE's security.

Sanctum The academic TEE Sanctum [103], designed for RISC-V, offers similar promises and a close programming paradigm compared to Intel SGX, by providing provable and robust software isolation, running in enclaves. The authors replaced Intel's opaque microcode/XuCode with two open-source components: the measurement root (mroot) and a secure monitor for verifiable protection. This solution adds a small set of minimally invasive hardware changes at the interfaces between generic building blocks to materialise the TEE hardware isolation, with minimal performance impacts. Besides, it includes a remote attestation protocol and a comprehensive design for deriving trust from a root of trust.

TIMBER-V The TEE TIMBER-V [196], also targeting RISC-V architectures, achieved the isolation of execution on small embedded processors thanks to hardware-assisted memory tagging. Tagged memory transparently associates blocks of memory with additional metadata. Unlike Sanctum, they aim to bring enclaves to smaller RISC-V featuring only limited physical memory. Similarly to TrustZone, the user mode (U-mode) and the supervisor mode (S-mode) are

split into a secure and normal world. The secure supervisor mode runs a trust manager (TagRoot), which manages the tagging of the memory. The secure user mode improves the model of TrustZone, as it can handle multiple concurrent enclaves, which are isolated from each other. They combine tagged memory with a memory protection unit (MPU) to support an arbitrary number of processes while avoiding the overhead of large tags. An MPU is suitable for systems with single address space implementations, as opposed to large systems that isolate processes in separate virtual address spaces using a memory management unit (MMU).

Intel TDX, Arm CCA, RISC-V CoVE, OpenPOWER PEF Intel Trust Domain Extensions (TDX) [107], Arm Confidential Compute Architecture (CCA) [108, 153], RISC-V Confidential VM Extension (CoVE) [197, 198] and OpenPOWER Protected Execution Facility (PEF) [199] are four emerging technologies that are VM-based TEEs, similar to AMD SEV-SNP. Intel TDX introduces the concept of Trust Domains (TDs), which are secure execution environments for VMs. The Secure-Arbitration Mode (SEAM) is a new CPU mode that enforces hardware isolation between TDs and untrusted system software. On the other hand, Arm CCA introduces Realms, which are isolated execution environments for VMs. The Realm Management Monitor (RMM) is a trusted firmware component responsible for managing Realms and enforcing hardware isolation between them and the untrusted hypervisor. Besides, RISC-V CoVE is an ongoing initiative that provides higher abstraction over the PMP instructions, introducing trusted virtual machines (TVMs) as secure execution environments. The TEE security manager (TSM) is a software module that enforces TEE security guarantees on a platform. It acts as the trusted intermediary between the hypervisor and the TVMs. Finally, PEF targets the general-purpose OpenPOWER computing platform, developed by IBM for its POWER processors. Upon instantiation, PEF checks the integrity of its secure virtual machines (SVMs) by leveraging the TPM, secure boot, and trusted boot. Additionally, PEF introduces a new most-privileged processor state called the *Ultravisor* to configure the hardware-based TEE. All these technologies leverage memory encryption and attestation mechanisms to ensure the confidentiality and integrity of VM code and data. As Intel TDX, Arm CCA and RISC-V CoVE provide security properties and abstractions comparable to AMD SEV-SNP, they are not further elaborated in this chapter.

Finally, some other emerging TEEs leveraging RISC-V are omitted as they lack remote attestation mechanisms, but are mentioned here for completeness. These technologies are yet to be researched for bringing such features. SiFive, a provider of commercial RISC-V processor SoCs, proposes Hex-Five MultiZone [200], a zero-trust computing architecture enabling the isolation of software, called *zones*. The multi-zone kernel ensures the sane state of the system using secure boot and the PMP instructions. HECTOR-V [201] is a design for developing hardened TEEs with a reduced TCB. Thanks to a tight coupling of the TEE and the SoC, the authors provide runtime and peripherals services directly from the hardware and leverage a dedicated processor and a hardware-based security monitor, which ensure the isolation and the control-flow integrity of the trusted applications, called *trustlets*. Lindemer *et al.* [202] enable simultaneous thread isolation and TEE separation on devices with a flat address space using an MPU, thanks to minor changes in the PMP specifications.

2.2.5 A summary of the TEE characteristics

Table 2.1 presents a summary of the key characteristics of the TEEs previously analysed in this chapter. Each feature can either be missing (○), partially (◑), or fully (●) available. This comparison focuses solely on the features of the TEEs, without considering supplementary security mechanisms, such as secure boot, that can improve their security guarantees.

	Features	SGX		TrustZone		SEV			RISC-V		
		Client SGX	Scalable SGX	TrustZone-A	TrustZone-M	Vanilla	SEV-ES	SEV-SNP	Keystone	Sanctum	TIMBER-V
Software	Encryption	●	●	○	○	●	●	●	●	○	○
Software	Integrity	●	●	○	○	○	○	●	●	○	○
Software	Freshness	●	●	○	○	○	○	●	●	○	○
Hardware	Encryption	●	●	○	○	●	●	●	●	○	○
Hardware	Integrity	●	○	○	○	○	○	○	●	○	○
Hardware	Freshness	●	○	○	○	○	○	○	●	○	○
Open source		◑	◑	◑	◑	◑	◑	◑	●	●	●
Industrial TEE		●	●	●	●	●	●	●	○	○	○
Number of concurrent domains		∞	∞	1	1	15	253/509/1006		Not evaluated		
Isolation granularity		Process-based		Partition-based		VM-based			Partition-based	Process-based	
System support for isolation		μcode + XuCode		SMC	MPU	Firmware			SMC + PMP		Tag + MPU

Table 2.1: The comparison of the state-of-the-art industrial and academic TEEs.

2.3 Are TEEs the silver bullet for security?

Despite the powerful security features of TEEs, it is essential to recognise that they are not a perfect solution. Like any security tool, TEEs have their limitations and potential vulnerabilities that must be carefully considered when designing and implementing trusted applications. Researchers around the world continuously identify and address vulnerabilities in TEEs, as is common practice for all security solutions. This section aims to provide a balanced perspective on the downsides and challenges associated with TEEs.

Code vulnerabilities One important vector of attacks targeting TEEs is vulnerable TA code. Poorly programmed TAs may inadvertently expose sensitive data or enable further exploitation of additional vulnerabilities, potentially compromising the integrity of the TEE runtime or trusted OS itself [203, 204]. Due to the privileged nature of these systems, a compromised TEE can grant attackers full control over the device. As this dissertation proposes trusted Wasm runtimes, such systems leverage software-based sandboxing techniques that can help mitigate the impact of a compromised TA, preventing it from taking down the entire TEE runtime.

Side-channel attacks A critical concern with TEEs is side-channel attacks. Side-channel attacks exploit unintended information leakage from the physical implementation of a system, such as timing information or power consumption. Attackers can infer sensitive data or cryptographic keys by analysing these side-channels, compromising the security features of these trusted environments. Researchers have demonstrated that all well-established TEEs are vulnerable to various side-channel attacks, such as Intel SGX [205], Arm TrustZone [204] and AMD SEV [206–211]. In response to these threats, CPU manufacturers typically provide security patches in a timely manner. However, as with all security solutions, this remains an ongoing cat-and-mouse game between attackers and defenders.

Programming constraints TEE development often requires adopting a different programming paradigm with various constraints. For example, OS services may be unavailable within the TEE, requiring developers to use an alternative API or implement their own services. Additionally, TEE development is often tied to a specific programming language, limiting flexibility and portability. Moreover, using a large TCB within the TEE increases the likelihood of vulnerabilities due to the increased attack surface, which is typically the case with the VM-based TEEs. This dissertation aims to address these challenges by proposing solutions that abstract the complexity of TEE development and provide a more secure and portable programming model.

In conclusion, while TEEs offer strong security guarantees, they should be considered as part of a defense-in-depth strategy rather than a standalone solution. Defense-in-depth is a security approach that leverages multiple layers of security to create a more resilient security posture. By combining TEEs with other security measures, organisations can better mitigate the risks associated with TEEs and build more trustworthy systems.

TEEs offer robust mechanisms for isolating sensitive workloads However, to fully leverage the security benefits of TEEs, one must ensure that the execution environment is genuine and not emulated or running on compromised hardware. This is where attestation comes into play as an essential part of most TEE implementations. Attestation offers strong assurances to both local and remote parties that the TEE's TCB is trustworthy and has not been tampered with. The following chapter is dedicated to better understanding attestation and examines how TEE implementations integrate this security mechanism to establish trust.

Chapter 3

Attestation in confidential computing

Attestation is a core mechanism that works in tandem with trusted execution environments (TEEs) to ensure trustworthy execution in untrusted environments. This chapter explores attestation standards, focusing on the IETF RATS architecture, and examines state-of-the-art industrial attestation solutions widely available for organisations and researchers, such as Intel SGX and AMD SEV. The analysis of these vendor-specific attestation schemes highlights the need for an agnostic attestation abstraction to enable interoperability between TEEs from different CPU manufacturers and developing portable trusted applications.

Chapter outline

3.1 Bringing trustworthy execution with TEEs . . . 40
3.2 Architecture and terminology . . . 40
3.3 Attestation types . . . 44
3.4 Overview of attestation for Intel SGX . . . 45
3.4.1 Local attestation . . . 45
3.4.2 Remote attestation: Enhanced Privacy ID (EPID) . . . 46
3.4.3 Remote attestation: Data Center Attestation Primitives (DCAP) . . . 48
3.5 Overview of attestation for AMD SEV . . . 50
3.5.1 Attestation with AMD SEV and SEV-ES . . . 50
3.5.2 Attestation with AMD SEV-SNP . . . 51
3.6 Portraying the need for an agnostic attestation abstraction . . . 52

3.1 Bringing trustworthy execution with TEEs

TEEs have emerged as a fundamental technology for enabling confidential computing, addressing the growing need for data confidentiality and secure code execution. TEEs provide hardware-enforced isolation and protection mechanisms, ensuring that sensitive data and applications remain secure even in the presence of compromised or malicious systems and actors. In the context of TEEs, three primary entities are typically involved: the data and application owners, the infrastructure providers hosting the TEE-enabled hardware, and the CPU manufacturers responsible for developing the TEE technology.

Despite the strong security guarantees offered by TEEs, confidential computing becomes truly meaningful when performing computations on sensitive data. Organisations must be able to trust that their data and applications are handled securely within genuine and uncompromised TEE instances. Attestation is the cornerstone principle that addresses this challenge by enabling TEEs to prove their trustworthiness to remote parties, assuring that the environment has not been tampered with and is running the expected software. Attestation mechanisms leverage cryptographic techniques to generate unforgeable proofs of the TEE's state, including measurements of the loaded code and the platform's configuration. Organisations can make informed decisions about the trustworthiness of the TEE instances they interact with by appraising these attestation reports, which later enable them to securely communicate with these trusted environments to transfer sensitive data and perform secure computations.

This chapter explores attestation using the IETF Remote Attestation Procedures (RATS) standard as a basis to formalise its application with TEEs. RATS provides an abstract architecture for describing attestation systems. Building upon this standard, we examine how attestation can be used in two widely used TEE solutions: Intel SGX and AMD SEV.

3.2 Architecture and terminology

The IETF RATS architecture [150] provides an abstract model for attestation that aims to establish common terminology and enable interoperability across heterogeneous systems. Research work has linked RATS to TEE attestation procedures, demonstrating its applicability in the context of confidential computing [212, 213]. RATS defines a set of roles, artefacts and interactions to design the creation, exchange, and verification environments to support trust decisions in various use cases, such as establishing the trustworthiness of TEE-protected workloads. Figure 3.1 illustrates the various components of the RATS architecture, which are further explained in this section with an improved mapping to TEEs compared to the IETF document. The key roles and entities defined in the RATS architecture are detailed below.

- **Attester**: A role performed by an entity (typically a device) whose evidence must be appraised to determine its trustworthiness. The attester produces evidence about itself. *Example with TEEs:* the attester can be (a part of) the trusted computing base (TCB) and the trusted application (TA) that generates evidence about the TEE's state and the workloads running within it.

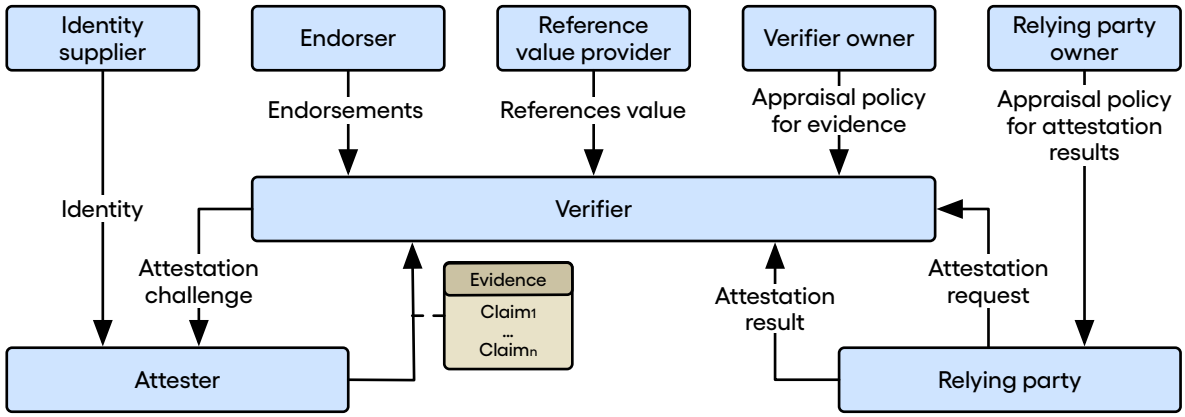


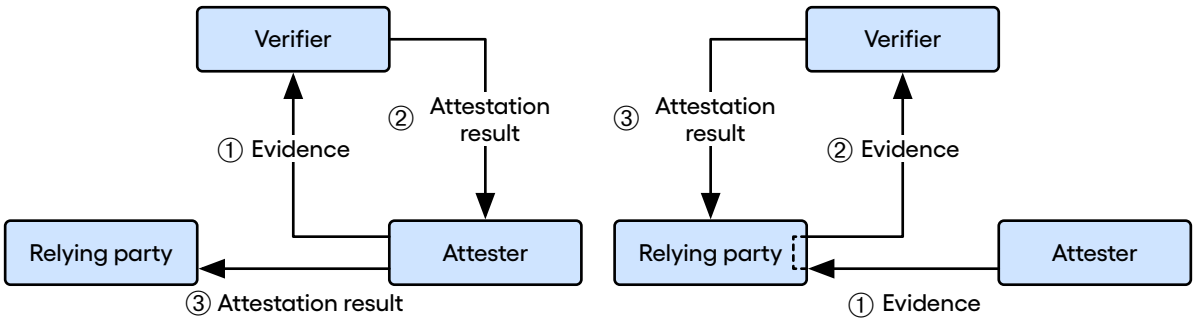
Figure 3.1: The RATS architecture, depicting the data flow of artefacts between the different entities. The diagram is extended by Sardar *et al*. and Sardar *et al*. with additional components for better representing TEEs attestation (described below) [212, 213].

- **Relying Party**: An entity that depends on the validity of information about an attester to make application-specific trust decisions. The relying party uses the appraisal policy for attestation results to evaluate the attestation results. *Example with TEEs:* a relying party could be a cloud service provider that needs to verify the integrity of a tenant's TEE-protected workload before allowing it to access sensitive data.
- **Verifier**: An entity that appraises the validity of evidence about an attester and produces attestation results to be used by a relying party. The verifier uses reference values, endorsements, and appraisal policies to assess evidence. *Example with TEEs:* the verifier could be a service provided by a CPU manufacturer that checks the validity of TEE-generated evidence against reference values and CPU manufacturer endorsements.
- **Endorser**: A role typically performed by a manufacturer whose endorsements help verifiers appraise the authenticity of evidence and infer the trust of the attester. *Example with TEEs:* the endorser could be the CPU manufacturer that provides endorsements vouching for the security properties and correct implementation of their respective TEEs.
- **Reference Value Provider**: An entity that provides reference values to help verifiers determine if claims in evidence are acceptable. *Example with TEEs:* the reference value provider could be the TEE manufacturer and TA owner of a TEE-protected workload, specifying the expected measurements of the TEE and the workload to ensure their integrity.
- **Verifier Owner**: An entity authorised to configure the appraisal policy for evidence in a verifier. *Example with TEEs:* the verifier owner could be the organisation deploying the verifier service, defining the policies for assessing TEE-generated evidence.
- **Relying Party Owner**: An entity authorised to configure the appraisal policy for attestation results in a relying party. *Example with TEEs:* the relying party owner could be the TA owner that sets the policies for how attestation results should be used in making trust decisions about TEE-protected workloads.

- **Identity Supplier**: A role proposed by Sardar *et al*. [213] as an extension to the initial RATS architecture. This role provides the attester with its cryptographic identity. *Example with TEEs:* the identity supplier could be the CPU manufacturer that provisions unique cryptographic materials during manufacturing.

The artefacts exchanged between RATS roles are detailed below. Similarly to the roles and entities, these data structures are mapped to their representation in TEEs.

- **Claim**: An asserted piece of information, typically in the form of a name/value pair, that comprises the structure of evidence. *Example with TEEs:* claims could include measurements of the TEE firmware, the workload code, and the TEE's security properties.
- **Evidence**: A set of claims generated and cryptographically authenticated or signed by an attester to be appraised by a verifier. *Example with TEEs:* evidence could consist of signed measurements of the TEE's state, the loaded workload, and any relevant configuration.
- **Endorsement**: A secure statement from an endorser vouching for the integrity of an attester's features, such as claims collection and evidence signing. *Example with TEEs:* an endorsement could be a certificate from the CPU manufacturer attesting to the security properties and correct implementation of the TEE.
- **Reference Value**: Known-good values for claims that a verifier compares against evidence as part of applying an appraisal policy for evidence. *Example with TEEs:* reference values could include expected measurements of the TEE firmware and workload code, ensuring their integrity.
- **Identity**: An artefact proposed by Sardar *et al*. [213] as an extension to the initial RATS architecture. The identity is a unique cryptographic identity associated with an attester. *Example with TEEs:* the identity could be a unique key pair provisioned by the CPU vendor during manufacturing, which is fused in the processor's die.
- **Attestation Request**: An artefact proposed by Niemi *et al*. [212] as an extension to the initial RATS architecture. This message is initiated by the relying party to start the attestation process. A few topological patterns instruct the flow of the attestation request, further elaborated in *topological patterns*. *Example with TEEs:* the relying party could send an attestation request to an attester, triggering the generation of evidence.
- **Attestation Challenge**: An artefact proposed by Niemi *et al*. [212] as an extension to the initial RATS architecture. This message is used to establish the freshness of evidence, typically containing a nonce generated by the verifier. *Example with TEEs:* the attestation challenge could include a nonce that the TEE must sign along with its evidence to prove the evidence is fresh and not a replay of previous attestation requests.
- **Attestation Result**: The output generated by a verifier, including information about an attester, where the verifier vouches for the validity of the results. *Example with TEEs:* the attestation result could indicate whether the TEE and its workload are trustworthy based on the appraisal of the submitted evidence by the CPU manufacturer.



(a) The RATS password model.

(b) The RATS background check model.

Figure 3.2: The topological patterns illustrate the attestation flow between the RATS entities.

- **Appraisal Policy for Evidence**: A set of rules that a verifier uses to evaluate the validity of evidence. *Example with TEEs:* the appraisal policy for evidence could specify the required measurements and properties for a TEE to be considered trustworthy.
- **Appraisal Policy for Attestation Results**: A set of rules that a relying party uses to instruct how it uses attestation results. *Example with TEEs:* the appraisal policy for attestation results could define how different attestation results should be interpreted and used in making trust decisions.

Topological patterns The RATS architecture defines topological patterns that illustrate the interaction flow between the attester, verifier, and relying party. The *passport model*, depicted in Figure 3.2a, is one such topological pattern, where the attester conveys evidence to the verifier for appraisal (①). The verifier compares the evidence against its appraisal policy and returns an attestation result to the attester (②). The attester treats the attestation result as opaque data and presents it to the relying party (③). The relying party then compares this information against its appraisal policy to make trust decisions.

In contrast to the passport model, the *background-check model*, shown in Figure 3.2b, involves the attester conveying evidence directly to the relying party (①), which treats it as opaque and forwards it to the verifier (②). The verifier appraises the evidence against its policy and returns an attestation result to the relying party (③), which evaluates it against its policy. In this model, the attester sends evidence to the relying party instead of an attestation result.

Attester composition An attester in the RATS architecture consists of at least one *attesting environment* and at least one *target environment*, as illustrated in Figure 3.3. The attesting environment is responsible for collecting claims about the target environment, such as taking measurements of code, memory, or other security-relevant assets (①). The attesting environment then formats these claims and uses cryptographic operations to generate evidence, later sent to the verifier (②). When applied to TEEs, the target environment may be the TA running within the TEE, while the TEE's TCB acts as the attesting environment. The composition supports a range of attester implementations, ranging from single-layer designs to more complex, layered attestation approaches where multiple attesting and target environments are nested.

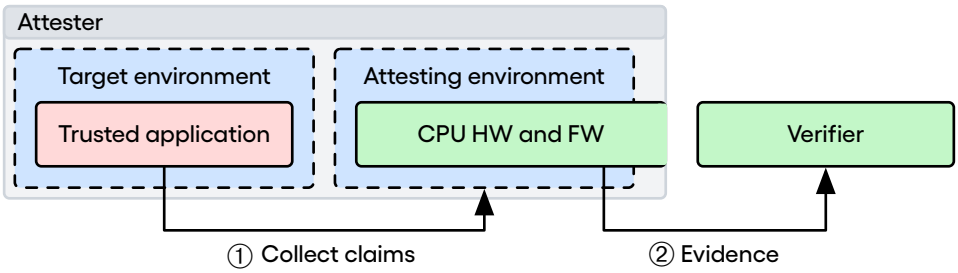


Figure 3.3: The attester owns an attesting environment that collects claims about a target environment. In the context of TEEs attestation, the target environment contains a trusted application, while the attesting environment comprises relevant CPU hardware and firmware parts. The green boxes (▢) represent the TCB of a relying party, whereas the red box (▢) indicates the component awaiting attestation.

3.3 Attestation types

Attestation plays a crucial role in establishing trust between different systems. This dissertation focuses on three types of attestation: local attestation, remote attestation, and mutual attestation, each serving a specific purpose in ensuring the trustworthiness of the involved parties.

Local attestation Local attestation enables an attester executing in a trusted environment to prove its trustworthiness to another trusted environment, which acts as both a verifier and a relying party. The key characteristic of local attestation is that the attester and verifier share the same TCB and are colocated within the same system, or on the same CPU if the attestation secret is bound to the processor. This shared TCB allows the attesting environment and the verifier to agree on common information. As a result, evidence in local attestation is typically secured using symmetric cryptography and authenticated with a message authentication code (MAC), leveraging a common secret between the attesting environment and the verifier.

Remote attestation In contrast to local attestation, remote attestation establishes trust between an attester and a verifier/relying party located on different systems, where the attester's TCB is no longer shared with the verifier. Therefore, the attesting environment relies on public-key cryptography to sign evidence, as the remote verifier cannot agree on a common secret beforehand. State-of-the-art TEEs often use a public key infrastructure (PKI) established by CPU manufacturers to infer trust from cryptographically signed evidence [185, 214, 215].

Remote attestation has been extensively researched in academia and industry, with various proposed approaches, including software-based, hardware-based, and hybrid (combining software and hardware) methods. Software-based remote attestation [216–218] is independent of specific hardware and well-suited for low-cost use cases. Hardware-based remote attestation, on the other hand, relies on a hardware-rooted *root of trust* to ensure the trustworthiness of the claims. This root of trust can be implemented using tamper-resistant hardware like Trusted Platform Modules (TPMs) [219], physical unclonable functions (PUFs) that prevent impersonation using unique hardware marks generated during manufacturing [220, 221], or hardware secrets fused into a die (*e.g.*, in the processor) and exclusively accessible to a trusted environment.

Mutual attestation Mutual attestation extends attestation by ensuring that both devices, which are part of the protocol, are attested, providing strong trust assurances in scenarios such as retrieving sensitive data from a sensing IoT device. In mutual attestation, both trusted environments authenticate each other, verifying that neither end has been tainted. Specialised protocols are designed to handle mutual attestation efficiently, reducing the number of interactions between the two peers compared to performing two separate attestation processes. Mutual attestation has notably been investigated in the context of TEEs [127, 222–224].

In the following sections, we analyse the attestation mechanisms used by several well-established industrial TEE implementations, namely Intel SGX and AMD SEV.

3.4 Overview of attestation for Intel SGX

Intel designed SGX to support local and remote attestation [122]. For remote attestation, Intel SGX offers two options: EPID [214] and DCAP [215]. EPID is an anonymous group signature scheme that allows an SGX enclave to prove its identity without revealing the specific platform it is running on. In contrast, DCAP uses the elliptic curve digital signature algorithm (ECDSA) for signing evidence, which allows the verification of them by any third party without the need for a dedicated attestation service from Intel. This enables more flexibility and control for organisations deploying SGX solutions in their own data centres or cloud environments.

3.4.1 Local attestation

Local attestation enables an enclave to prove its trustworthiness to another enclave running on the same platform. Intel SGX provides two instructions to support local attestation: `EREPORT` and `EGETKEY`. The `EREPORT` instruction generates an authenticated evidence, called *report* in SGX jargon, containing the enclave's measurement along with other relevant security parameters, while the `EGETKEY` instruction allows the enclave to act as the relying party to retrieve the report key necessary to verify the authenticity of the evidence. The evidence issued by `EREPORT` only conveys trust when used on the same platform, as it is not signed by Intel's attestation PKI.

Measurement is an essential concept in SGX attestation, as it provides a cryptographic representation of the enclave's identity. There are two measurement registers: `MRENCLAVE` and `MRSIGNER`. `MRENCLAVE` is a hash of the enclave's code, data, and configuration, ensuring that any changes to the enclave lead to a different measurement value. `MRSIGNER`, on the other hand, represents the identity of the enclave's signer, like the enclave owner or a trusted authority.

The local attestation workflow, also known as the *intra-platform enclave attestation* process, involves a sequence of steps that enables one enclave (Enclave A), the attester, to prove its trustworthiness to another enclave (Enclave B), the relying party and verifier, on the same platform. Figure 3.4 illustrates this process, which is further elaborated below.

1. Enclave A obtains Enclave B's `MRENCLAVE` (verifier's identity) value through an untrusted communication channel.

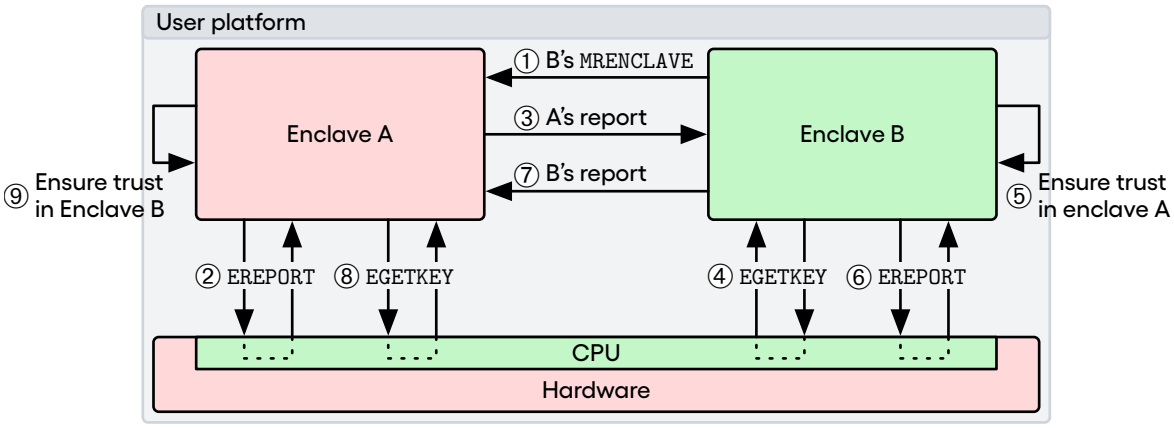


Figure 3.4: The local attestation flow of Intel SGX. The green boxes (▢) are the TCB from Enclave B's standpoint, while the red boxes (▢) are untrusted.

2. Enclave A invokes `EREPORT` instruction with Enclave B's `MRENCLAVE` to generate an authenticated report (evidence), specifically tailored for Enclave B.
3. Enclave A sends the report to Enclave B via the untrusted channel.
4. Enclave B retrieves its report key using `EGETKEY` instruction and verifies the A's report authenticity by recomputing the MAC and comparing it with the one in the A's report.
5. If the MAC matches, Enclave B compares Enclave A's measurement (`MRENCLAVE`) in A's report against a reference value to establish trust in Enclave A (part of the attester).
6. Enclave B reciprocates by generating a report for Enclave A by invoking `EREPORT` instruction with Enclave A's `MRENCLAVE` from A's received report.
7. Enclave B sends its report to Enclave A.
8. Enclave A retrieves its report key using `EGETKEY` instruction and verifies the B's report authenticity by recomputing the MAC and comparing it with the one in the B's report.
9. If the MAC matches, Enclave A compares Enclave B's measurement (`MRENCLAVE`) in B's report against a reference value to establish trust in Enclave B.

Local attestation is necessary when communicating with enclaves on the same platform to establish trusted communication paths. To enable trust with remote platforms, Intel SGX integrates two attestation schemes: Intel SGX EPID and DCAP, detailed in the following sections.

3.4.2 Remote attestation: Enhanced Privacy ID (EPID)

Intel introduced the Enhanced Privacy ID (EPID) scheme to address privacy concerns associated with remote attestation using standard asymmetric signing schemes. EPID is an anonymous group signature scheme that extends the direct anonymous attestation (DAA) algorithm [225] and allows an Intel SGX enclave to prove its identity and trustworthiness without revealing the

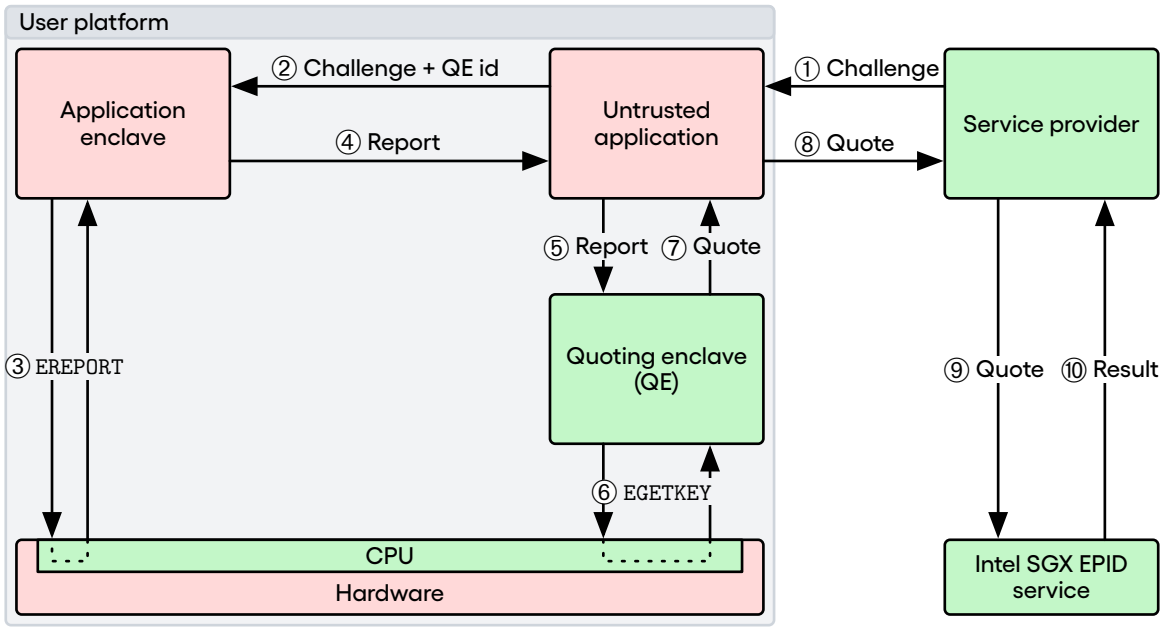


Figure 3.5: The remote attestation flow of Intel SGX EPID. The green boxes (▢) are the TCB from the service provider standpoint. The red boxes (▢) are untrusted.

specific platform. In EPID, each signer belongs to a group, and verifiers use the group's public key to verify signatures. The scheme is designed to provide anonymity, as the verifier cannot associate a given signature with a particular attester of the group.

The remote attestation process for Intel SGX, also known as the *inter-platform enclave attestation* process, involves the Quoting Enclave (QE), a special enclave from Intel devoted to remote attestation and considered in SGX's TCB. The QE verifies the report generated by the application enclave using the local attestation process (through `EREPORT` instruction), replaces the MAC in the report with an EPID signature and adds security-related data, such as information about the signing QE, creating evidence for external systems, called the *quote* structure. A remote party can verify the quote using the EPID group public key to establish trust in the application enclave. The remote attestation process, illustrated in Figure 3.5, is summarised below.

1. The service provider (relying party) or a delegated verifier sends an attestation challenge to prove that the enclave application and the SGX's TCB (attester) is a genuine platform.
2. The untrusted side of the enclave application, named *untrusted application*, forwards the challenge to the enclave application, along with the QE's identity (`MRENCLAVE`).
3. The enclave application generates the attestation report with the QE's identity using the `EREPORT` instruction, which contains the challenge and a public key for establishing a secure communication channel upon a successful attestation.
4. The enclave application sends the report to the untrusted application.
5. The untrusted application forwards the report of the enclave application to the QE.

6. The QE retrieves its report key using the EGETKEY instruction and uses it to verify the report of the enclave application, hence performing a local attestation.
7. Upon successful appraisal, the QE signs the report with its EPID key and returns the resulting quote structure (evidence) to the untrusted application.
8. The untrusted application sends the quote to the service provider (relying party).
9. The service provider delegates the verification of the quote to the Intel Attestation Service (IAS) acting as the verifier, compares the enclave's measurement in the quote against expected reference values to establish trust in the enclave and SGX TCB (attester).
10. If the quote is deemed trustworthy, the IAS informs the service provider to operate with this particular enclave instance.

Intel has recently announced the end-of-life for its EPID-based IAS, scheduled for April 2025, as the company shifts its focus towards the more flexible Intel SGX DCAP (described below) and the newly introduced Intel Trust Authority, a zero-trust attestation software-as-a-service [226].

3.4.3 Remote attestation: Data Center Attestation Primitives (DCAP)

Intel introduced the Data Center Attestation Primitives (DCAP) scheme to support non-Intel attestation infrastructures for Intel SGX. DCAP allows third parties to author their own SGX attestation infrastructure, providing flexibility for use cases where relying on Intel's services is not feasible or desirable. The foundation of DCAP is the Intel-provided Provisioning Certification Enclave (PCE), which acts as a local certificate authority (CA) for the Quoting Enclave (QE) running on the same platform. The PCE uses a unique, hardware-based Provisioning Certification Key (PCK) to sign certificates for the QE' attestation key. In contrast to the EPID-based attestation, where Intel manages the QE and its attestation key, DCAP enables third parties to develop their own QEs and controls the attestation key locally. The PCE certifies the attestation key, creating a certificate chain rooted in an Intel-issued certificate, which allows the quotes generated by a custom QE to be verified by a verifier without direct involvement from Intel.

Intel DCAP consists of an initialisation phase, executed once per TCB configuration, and an attestation phase. The two phases are depicted in Figure 3.6. The initialisation phase uses the PCE to certify the QE's public attestation key via local attestation, as outlined below.

(i) The QE invokes the EREPORT instruction to perform a local attestation with the PCE.

(ii) The QE sends the generated report with its public attestation key to the QE.

(iii) The PCE verifies the QE's report using its report key obtained via the EGETKEY instruction, completing the local attestation with the QE.

(iv) Upon a successful attestation, the PCE signs the QE's attestation public key using its PCK, creating a certificate-like structure rooted in an Intel-issued certificate, known as the *attestation key certificate*. This certificate is used to appraise attestation quotes issued by the QE during the attestation phase.

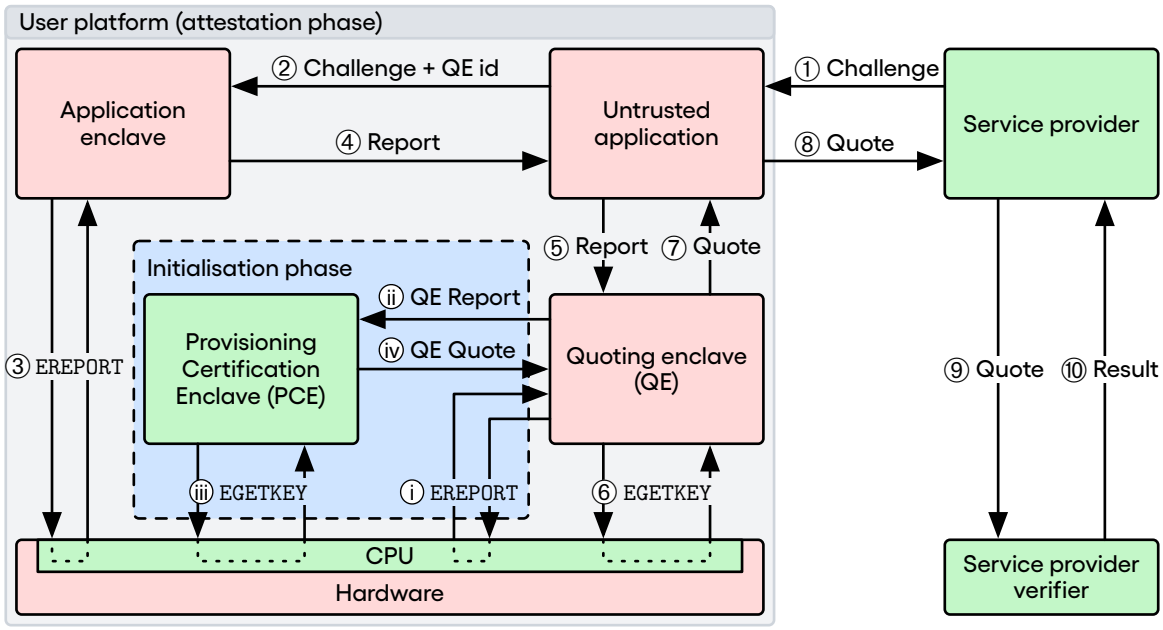


Figure 3.6: The remote attestation flow of Intel SGX DCAP. The green boxes (▢) are the TCB from the service provider standpoint. The red boxes (▢) are untrusted. The blue box (▢) denotes the initialisation phase. Unlike Intel EPID, the QE is untrusted, as potentially provided by a third party, and identified in the quote.

In the DCAP attestation phase, the verifier operates independently of Intel's online attestation services, following the steps 1–6 of the EPID-based attestation process.

1. The service provider (relying party) or a delegated verifier sends an attestation challenge to prove that the enclave application and the SGX's TCB (attester) is a genuine platform.
2. The untrusted side of the enclave application, named *untrusted application*, forwards the challenge to the enclave application, along with the QE's identity (`MRENCLAVE`).
3. The enclave application generates the attestation report with the QE's identity using the `EREPORT` instruction, which contains the challenge and a public key for establishing a secure communication channel upon a successful attestation.
4. The enclave application sends the report to the untrusted application.
5. The untrusted application forwards the report of the enclave application to the QE.
6. The QE retrieves its report key using the `EGETKEY` instruction and uses it to verify the report of the enclave application, hence performing a local attestation.
7. Upon successful appraisal, the QE creates the quote (evidence) based on the report, which is signed using the QE's private attestation key and returns to the untrusted application.
8. The untrusted application sends the quote to the service provider (relying party).

9. The service provider delegates to a verifier the verification of the signature of the quote using the QE's certified public key, checks the certificate chain up to the Intel Root certificate authority and ensures that the PCK and the platform TCB are not revoked.
10. The verifier then validates the enclave's response to the challenge, compares the enclave's measurement in the quote against the expected reference value to establish trust in the enclave and SGX TCB (attester) and informs the service provider.

3.5 Overview of attestation for AMD SEV

AMD designed SEV to support remote attestation, enabling a guest owner (relying party) to verify the trustworthiness of a VM running on an untrusted cloud provider's infrastructure (attester). While AMD SEV does not provide local attestation features, two VMs on the same system can still attest to each other using the remote attestation mechanism. AMD SEV and SEV-ES support boot-time attestation, where the attestation evidence is generated when the VM is launched [123, 182]. In contrast, AMD SEV-SNP introduces more flexible attestation features, allowing the guest owner to request attestation at any time during the VM's lifecycle [185].

3.5.1 Attestation with AMD SEV and SEV-ES

In AMD SEV and SEV-ES, remote attestation is performed when a confidential VM is launched on the cloud provider's platform. The primary goal of this attestation process is to provide the guest owner with evidence that the VM has been securely instantiated on a genuine AMD platform and that its initial state is trustworthy. The guest owner communicates with the SEV firmware through a secure channel, which is used to transmit the attestation evidence. If the guest owner deems the AMD platform trustworthy based on the appraisal of the evidence, they can provision sensitive data to the VM, such as disk decryption keys and sensitive data. The remote attestation workflow for launching a confidential VM is illustrated in Figure 3.7 and can be summarised below.

1. The service provider (relying party) or a delegated verifier deploys the VM, including an encrypted disk image and an attestation challenge, to the cloud provider.
2. The hypervisor prepares the launch of the guest VM and invokes the SEV firmware to encrypt the VM's memory using the `LAUNCH_START` command, which creates a guest context with the service provider's public key.
3. The hypervisor loads the guest VM into memory using the `LAUNCH_UPDATE_DATA` and `LAUNCH_UPDATE_VMSA` command to encrypt the memory pages and compute the measurement of the initial VM state.
4. Once the VM is fully loaded, the hypervisor issues the `LAUNCH_MEASURE` command, which instructs the SEV firmware to generate an attestation report (evidence) containing the measurement, SEV version and the challenge, among other security-related information.
5. The SEV firmware sends the attestation report to the service provider (relying party).

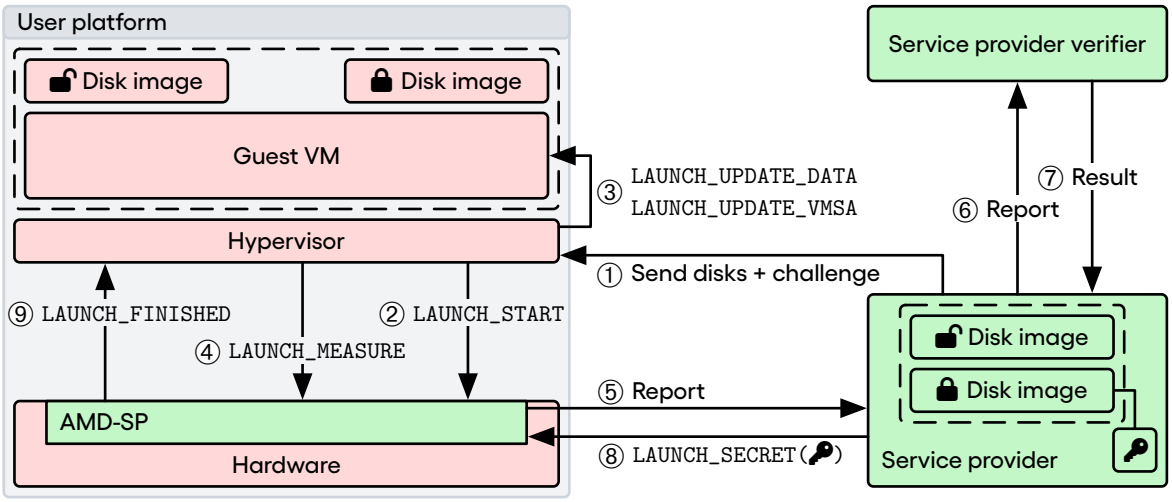


Figure 3.7: The remote attestation flow of AMD SEV and AMD SEV-ES during guest VM launch. The green boxes (▢) are the TCB from the service provider standpoint. The red boxes (▢) are untrusted.

6. The service provider delegates to a verifier the appraisal of the attestation report by validating the challenge, the VM's measurement and the other security-related data against reference values to establish trust in the launched VM and the AMD TCB (attester).
7. If the attestation report is deemed trustworthy, the verifier informs the service provider to operate with this particular guest VM instance.
8. The service provider provisions sensitive data, such as disk encryption keys, using the `LAUNCH_SECRET` command.
9. The launch process is finalised by executing the `LAUNCH_FINISHED` command, indicating that the VM is ready to start execution.

The verifier can operate independently of AMD during attestation, as long as it has been appropriately provisioned with AMD-issued certificates beforehand, similarly to Intel SGX DCAP.

3.5.2 Attestation with AMD SEV-SNP

AMD designed SEV-SNP to support remote attestation during guest VM runtime, enabling the guest VM to request attestation reports at any time, in addition to the boot-time attestation from the previous SEV versions. This flexible attestation mechanism allows the guest VM to interact directly with the AMD Secure Processor (AMD-SP) through a protected communication path established during the VM launch process. The remote attestation process in AMD SEV-SNP, depicted in Figure 3.8, follows these steps:

1. The service provider (relying party) or a delegated verifier sends an attestation challenge to the guest VM to verify the authenticity of the VM and the SEV TCB (attester) as a genuine AMD platform.

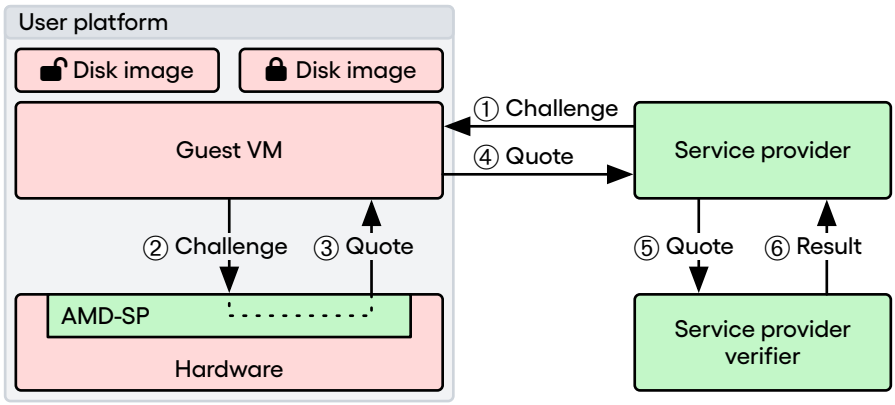


Figure 3.8: The remote attestation flow of AMD SEV-SNP during guest VM runtime. The green boxes (▢) are the TCB from the service provider standpoint. The red boxes (▢) are untrusted.

2. The guest VM requests an attestation report (evidence) from the AMD-SP, which includes the challenge and other security-related assets, such as a public key for establishing a secure communication channel upon a successful attestation.
3. The AMD-SP issues the attestation report, which is signed by the Versioned Chip Endorsement Key (VCEK), proving both the authenticity of the platform and the TCB version.
4. The attestation report is forwarded to the service provider (relying party).
5. The service provider delegates to a verifier the appraisal of the attestation report, which validates the challenge, the VM's measurement, and the other security-related information against reference values to establish trust in the guest VM and the AMD TCB (attester).
6. If the attestation report is deemed trustworthy, the verifier informs the service provider to operate with this particular guest VM instance.

This attestation scheme allows third-party services and guest owners to verify the trustworthiness of the guest VM at runtime, offering attestation features similar to Intel SGX DCAP.

3.6 Portraying the need for an agnostic attestation abstraction

The lack of standardised attestation protocols for TEEs has led to a fragmented landscape, with CPU vendors using ad-hoc solutions tailored to their specific hardware implementations. Intel SGX and AMD SEV use different attestation mechanisms. Intel SGX supports the EPID and DCAP schemes, while AMD SEV relies on a secure channel between the guest owner and the SEV firmware for boot-time attestation, with SEV-SNP introducing more flexible runtime attestation features. This diversity in attestation schemes complicates the development of TAs, as developers must understand and rely on vendor-specific API and security considerations.

In academia, numerous research efforts have focused on bringing attestation to Arm TrustZone, which lacks built-in attestation mechanisms. However, these solutions often introduce ad-hoc

attestation protocols, further contributing to fragmentation. While recent initiatives like the TLS attestation draft of the CCC Attestation Special Interest Group have emerged to address this challenge by developing modern, unified solutions for handling attestation independently of the underlying TEE implementation, these efforts are still in the early stages of development and require further research to reach maturity and widespread adoption [144].

Open-source attestation tools for multiple TEEs, such as OpenEnclave [135] and Librats [137], offer some degrees of abstraction and interoperability. However, their tight coupling with specific programming paradigms or languages limits their broader applicability. For instance, OpenEnclave and Librats are primarily designed for C and C++ development. This coupling between attestation tools and programming languages can hinder the adoption of these solutions in heterogeneous environments where multiple languages and frameworks are in use.

To tackle these challenges, we propose to reduce the complexity of TEE development and attestation by abstracting these concepts using a lightweight intermediate bytecode. This approach can offer a set of TEE primitives, including attestation, that can work across different CPU manufacturers, acting as a portable solution. For this task, we opted for WebAssembly (Wasm) as an intermediate language and a portable compilation target due to the many characteristics that make it suitable for embedding into constrained environments like TEEs. Wasm proposes lightweight specifications, which focus on building a minimal, performant and secure runtime environment, which reduces the TCB of TAs. It offers a simple system layer named WebAssembly System Interface (WASI), which can be repurposed to serve trusted TEE API or ensure that no confidential data is leaked through these controlled channels. Moreover, this solution can reuse open-source tools like Librats, benefiting from their established codebases and security features while presenting developers with a language-agnostic attestation API. Finally, Wasm is a compilation target for many modern programming languages, including C++, Rust, and Go, with many other higher-level languages like C# and Python beginning to integrate it.

The next chapter introduces WebAssembly and its many advantages as a solution for abstracting the complexity of writing and deploying trusted applications in TEEs.

Chapter 4

WebAssembly

WebAssembly (Wasm), a new standard and compilation target for many programming languages, has emerged as a versatile and high-performance technology for both web and standalone applications. This chapter explains the rationale behind the introduction of Wasm, examines its technical details and inner mechanisms, and explores the substantial advantages of this technology. Furthermore, it discusses the potential integration between Wasm and confidential computing through the use of trusted execution environments (TEEs).

Chapter outline

4.1 An agnostic standard for the cloud-edge continuum 56
4.2 WebAssembly under the hood 57
4.2.1 Bringing abstraction to programming languages 58
4.2.2 Bringing abstraction to system environment 58
4.2.3 Bringing abstraction to execution modes 60
4.2.4 Bringing abstraction to operating system services 61
4.3 A building block for interoperable TEE architectures 63
4.4 A rich ecosystem 65
4.4.1 An overview of the standalone runtimes landscape 65
4.4.2 A summary of runtimes characteristics 67
4.4.3 A balance of performance and lightweightness for TEEs 67

4.1 An agnostic standard for the cloud-edge continuum

The rapid evolution of computing infrastructure has led to the emergence of the *cloud-edge continuum*, a seamless integration of cloud, edge, and IoT devices. This paradigm shift is driven by the increasing demand for real-time data processing, reduced latency, and improved efficiency. The cloud-edge continuum enables applications to leverage the strengths of each computing environment, from the vast resources of the cloud to the low-latency processing of edge devices and the ubiquitous presence of IoT sensors. This interconnected ecosystem allows for more efficient and responsive services that answer the needs of modern, data-intensive applications. However, the heterogeneity of the cloud-edge continuum poses significant challenges in terms of software development, deployment, and security.

Prior to the advent of modern portable compilation targets, developers relied on platform-specific solutions or virtual machines to address the challenges of heterogeneous computing environments. The Java virtual machine (JVM) and Microsoft's Common Language Runtime (CLR) for .NET provided a degree of portability by allowing applications to be compiled once and executed on various platforms. However, these solutions often came with performance overheads and were limited to specific programming languages. Besides, Google introduced Native Client (NaCI) and Portable Native Client (PNaCl) to enable secure execution of native code in web browsers, but these technologies were tightly coupled to specific architectures and lacked the flexibility needed for the heterogeneous landscape of the cloud-edge continuum.

As the computing landscape evolves, the need for secure and privacy-preserving computation has become paramount. Confidential computing, which leverages TEEs to protect sensitive data and code, has emerged as a promising solution. Nonetheless, the current confidential computing ecosystem is fragmented, with each CPU vendor offering fundamentally different proprietary TEE implementations, such as Intel SGX, AMD SEV, and Arm TrustZone. This fragmentation hinders portable and secure application development, as developers must satisfy the constraints of each TEE architecture. Moreover, the tight coupling between TEEs and processor architectures limits the flexibility and scalability required for the cloud-edge continuum.

To address the challenges posed by the heterogeneous nature of the cloud-edge continuum and the limitations of current confidential computing solutions, the work described in this thesis pointed towards advocating the adoption of the recent open standard WebAssembly (Wasm) as a lightweight, efficient, and secure portable compilation target. Wasm's platform-agnostic design and sandbox make it an ideal candidate for bridging the gap between diverse computing environments while providing a robust foundation for secure execution. Developers can create portable and secure software that seamlessly runs across the cloud-edge continuum by compiling applications to Wasm, from resource-constrained IoT devices to powerful cloud servers. The combination of Wasm and TEEs establishes a mutually beneficial security model, protecting sensitive data and computations from unauthorised access while also safeguarding the underlying infrastructure from potentially malicious code. This synergy lays the foundation for a more trustworthy and collaborative confidential computing ecosystem, enabling the development of secure and privacy-preserving applications that span the cloud-edge continuum.

4.2 WebAssembly under the hood

WebAssembly (Wasm) [94, 227] is a novel, general-purpose virtual instruction set architecture (ISA) designed as a portable compilation target for modern programming languages like C, C++, Rust and Go. Developed through a collaborative effort by major technology companies such as Microsoft, Google, Apple and Mozilla, Wasm was initially proposed to enhance the performance of web applications. However, its design is not intrinsically tied to the web, making it equally suitable for standalone applications. Hence, the versatility of Wasm has led to its adoption in various domains beyond the web, such as serverless computing [228–231], edge computing [232, 233], and IoT devices [234–236]. Wasm is a platform-agnostic and efficient execution model, which makes it a de facto attractive choice as a unified execution technology for the cloud-edge continuum, spanning many computing environments from resource-constrained IoT devices to cloud servers. Wasm specifications promote the design goals as follows.

- **Safe**: Wasm's sandboxed execution environment and well-defined semantics ensure memory safety, making it suitable for untrusted code execution. Traditionally, managed language runtimes, such as JavaScript VMs, have been used to enforce memory safety and prevent programs from compromising user data or system state. Nonetheless, these runtimes have not adequately supported portable, low-level code, such as memory-unsafe compiled C/C++ applications that require fast execution without garbage collection. Wasm addresses this gap by providing a safe execution environment for low-level code.
- **Fast**: Wasm's low-level, compact binary format, similar to that generated by a C/C++ compiler, enables efficient execution, approaching near-native performance through optimised just-in-time (JIT) and ahead-of-time (AOT) compilation techniques. Wasm aims to minimise these overheads and provide near-native execution speed.
- **Portable**: Wasm's platform-independent design allows compiled Wasm modules to run consistently across different hardware architectures, operating systems, and environments, facilitating seamless deployment across the cloud-edge continuum.
- **Compact**: Wasm's binary format is designed to be compact, minimising storage space and load time overhead, which is particularly beneficial for resource-constrained edge devices and bandwidth-limited network scenarios. Traditionally, code on the web is transmitted as JavaScript source, which is significantly less compact than a binary format, even when minified and compressed. Wasm's binary format offers a more efficient alternative, reducing the size of transmitted code and improving performance.

The remainder of this chapter explores the various aspects of Wasm, including the CPU and memory representation, execution modes, and how it abstracts the underlying operating system and programming languages. Furthermore, it highlights the potential synergies between Wasm and TEEs and examines the currently available Wasm runtimes.

4.2.1 Bringing abstraction to programming languages

Wasm is designed as a portable compilation target for modern high-level programming languages such as C, C++, Rust, and Go. Developers can create applications that abstract away the underlying technological choices by compiling these software stacks to Wasm, enabling a more interoperable development ecosystem. Furthermore, support for additional programming languages is made possible by compiling their runtimes into Wasm, albeit at the cost of a double runtime execution overhead. One notable example is Blazor [237], which compiles Mono [238], a .NET runtime hosting, into a single Wasm binary to load and execute Common Intermediate Language (CIL) instructions directly from within the browser.

Wasm code is represented in two distinct formats: the human-readable textual format (WAT) and the compact Wasm binary format (bytecode). The textual format is a valuable representation for learning, testing, and debugging Wasm programs. In contrast, the binary format is optimised for efficient execution and transmission. These two formats are interconvertible using tools like the WebAssembly Binary Toolkit (WABT) [239], allowing developers to transition between them as needed. The Wasm bytecode is structured as a sequence of instructions that operate on a conceptual stack machine, with each instruction consuming its operands from the stack and pushing its results back onto it.

Most programming languages toolchains leverage the LLVM compiler infrastructure [240] when targeting Wasm. LLVM separates compiler front-ends, which convert high-level programming languages into an intermediate representation (IR), from compiler back-ends, which transform the IR into CPU-specific machine code. This modular architecture enables the compilation of source code in Wasm by leveraging existing LLVM compiler front-ends and only requiring the implementation of a back-end to translate the IR into Wasm bytecode. Furthermore, some programming language toolchains that do not typically rely on LLVM for compilation have found ways to integrate LLVM when Wasm is the compilation target, such as Go using TinyGo [241, 242]. Alternatively, other compilers use ad-hoc backend implementations to generate Wasm bytecode directly, as demonstrated by TeaVM, which compiles Java into Wasm [243].

Dedicated Wasm compilers, such as Emscripten [244] which relies on LLVM, typically generate two types of output artefacts: one tailored for web deployment and another for standalone execution. The web-oriented artefact includes the necessary JavaScript glue code to integrate Wasm binaries into web development workflows, thereby facilitating the embedding of Wasm code into web applications. Conversely, the standalone artefact produces a Wasm binary that can be executed independently of the web environment. Instead, it interacts with the operating system via the runtime (a topic explored in Section 4.2.4).

4.2.2 Bringing abstraction to system environment

Wasm is designed as a low-level, platform-independent virtual ISA that abstracts the underlying hardware processor. Wasm is a binary instruction format for a stack-based virtual machine with its own dedicated bytecode, allowing it to operate independently of the host machine's ISA, thus ensuring platform-agnostic execution. The bytecode consists of instructions that manipulate values on an operand stack. Each instruction consumes its operands from the stack and

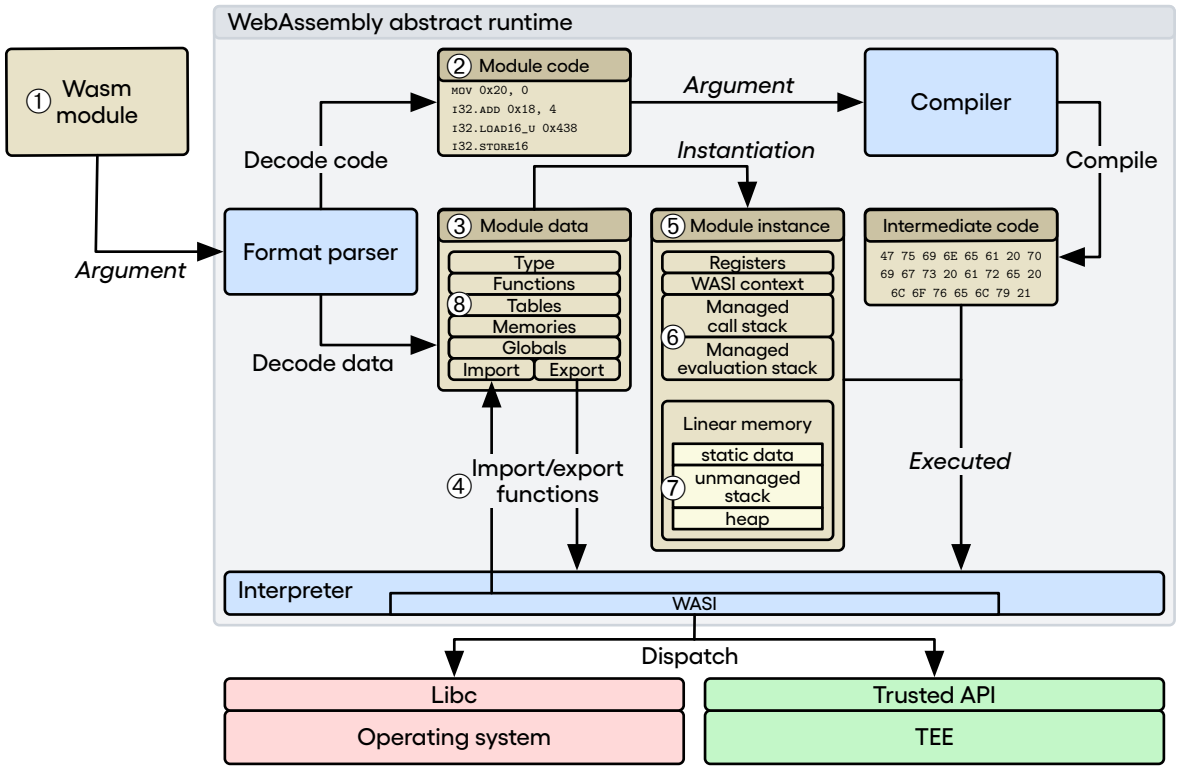


Figure 4.1: An abstract representation of a Wasm runtime. The blue boxes (▢) are components in the runtime, the brown boxes (▢) are data structures, the green boxes (▢) are trusted components (in a TEE), and the red boxes (▢) are untrusted components.

pushes its results back onto it, similarly to the CLR. Additionally, Wasm uses a linear memory model, further segmenting the memory layout and management from the underlying OS. Wasm is executed using a runtime, similarly to other intermediate languages like the .NET CIL. Figure 4.1 illustrates an abstract Wasm runtime, which comprises the minimum components for executing Wasm applications. Parts of Figure 4.1 are described in the following paragraphs.

Wasm modules Wasm programs are organised into *modules*, which serve as the deployment, loading, and compilation units (①). A module contains the code (②) and necessary data (③), such as the *types*, *functions*, *tables*, *memories*, and *globals*. Functions are the basic units of code in Wasm, defined within a module or imported from external sources like the runtime. Tables are resizable arrays of references that can hold function references, enabling dynamic dispatch and indirect function calls. Memories define the linear memory available to a Wasm module, which is a contiguous, mutable array of raw bytes. Globals are module-wide variables that any function within the module can access. A module can also declare imports and exports (④), which notably allows interfacing with the underlying system and exporting entry points. A *module instance* is the runtime representation of a compiled Wasm module (⑤). It is created when a Wasm module is instantiated and includes all the state and resources necessary to execute the module's code, such as the virtual registers, stacks, and heap. Each module instance maintains references to these entities so they can be accessed and manipulated during execution.

Memory representations Wasm distinguishes between managed and unmanaged data [245]. Managed data resides in dedicated stacks directly handled by the VM: the *managed call stack* stores the locals and return addresses, while the *managed evaluation stack* holds the intermediate values of instructions (⑥). Wasm code can only interact with managed data implicitly through instructions and cannot directly modify its underlying storage, protecting these structures against buffer overflow exploits. However, not all local variables are stored in the managed stacks, as Wasm specifications define only four primitive types (32 and 64 bit integers and floating-point data). Non-scalar data, such as strings and arrays, must be stored in the linear memory, which is a contiguous, untyped array of bytes. Also, any variable whose address is taken in the source program, such as out parameters, must be stored in the linear memory because managed data has no addressable location. The linear memory is the primary storage mechanism for Wasm applications and is separate from the code space, managed stacks, and runtime data structures. It is organised into pages, each with a fixed size of 64 KiB. The linear memory layout typically includes static data, an *unmanaged stack*, and a heap, which are handled by the compiler-generated code (⑦). The unmanaged stack stores local variables that cannot be held in the managed stacks. The heap is located at the end of the linear memory, allowing it to grow towards higher addresses when additional memory is requested from the Wasm application through the runtime. The static data section in the linear memory contains data that is valid for the entire lifetime of the program, such as statically initialised constants, global variables, and zero-byte ranges. The order of the unmanaged stack, heap, and static data in the linear memory depends on the specific compiler and its configuration.

Functions and tables In Wasm, functions are the primary unit of code execution. A *function instance* is a runtime representation of a function, acting as a closure over the original function defined in the module. It captures the runtime module instance, allowing it to resolve references to other definitions during execution. Wasm supports internal and external functions, where internal functions are defined within the module, and external functions are imported from the host environment or other modules. Functions can also be exported so other modules or the host environment can call these entry points to start code execution (④). Wasm also supports indirect function calls through the use of tables (⑧). A table is a resizable array of function references. An indirect call instruction takes an index into the table and invokes the corresponding function. This mechanism enables dynamic dispatch and supports function pointers in languages like C/C++. Notably, tables in Wasm help enforce control flow integrity by ensuring that function calls are made only to valid, predefined functions. This is achieved through type checking, where each indirect call checks the type of the function being called against the expected type. This prevents type confusion attacks where an attacker might try to call a function with an incompatible signature. Additionally, Wasm implementation check their bounds when table indices are accessed, ensuring that calls are made only within the valid range of the table. This mitigates buffer overflow attacks that could otherwise lead to arbitrary code execution.

4.2.3 Bringing abstraction to execution modes

As an intermediate representation, Wasm bytecode requires interpretation or compilation before execution. Wasm runtimes use three strategies for bytecode execution, detailed as follows.

- **Interpreter mode**: As the most direct strategy for running Wasm applications, interpreter mode offers a small footprint and low memory consumption at the cost of slower execution speed. This strategy can be implemented using a textbook approach, directly interpreting the Wasm bytecode, or optimised by converting the bytecode to an internal representation tailored for the interpreter. The latter approach improves performance by reducing the overhead of decoding Wasm instructions during execution, albeit with increased memory consumption due to the additional internal representation. While interpreter mode is the slowest of the three execution strategies, it provides a simple and memory-efficient option for executing Wasm applications, making it suitable for resource-constrained environments or platforms where dynamic loading of machine code is prohibited.
- **Just-in-time (JIT) mode**: JIT mode involves compiling the Wasm bytecode into machine code for the host machine's ISA at runtime. This strategy offers improved performance compared to interpreter mode by leveraging the optimisations and features of a JIT compiler. Two common approaches exist for implementing JIT mode in Wasm runtimes. The first approach uses an existing JIT compiler, such as LLVM [240], which offers the best performance due to its mature optimisations and extensive support for various architectures. However, integrating a full-fledged JIT compiler like LLVM into a Wasm runtime can significantly increase the runtime's size and compilation time. Moreover, satisfying the dependencies of a complex JIT compiler across some target host platforms can present additional challenges, particularly in platforms with limited support for the required libraries and toolchains. The second approach involves developing a custom JIT compiler tailored specifically for Wasm. While this requires more upfront development effort, as creating a JIT compiler is a complex task, it allows for fine-grained control over the compiler's size and dependencies, enabling optimisations specific to the target host's constraints.
- **Ahead-of-time (AOT) mode**: AOT mode offers the best performance among the three execution strategies by compiling the Wasm bytecode into machine code before deploying it to the target platform. This approach shifts the compilation overhead to an earlier stage, enabling the runtime to focus solely on executing the generated machine code, resulting in faster startup times and reduced memory footprint. However, AOT mode requires knowledge of the target machine's ISA at compile time, limiting the portability of the compiled Wasm application. Furthermore, the AOT compilation process must occur in a trusted environment to maintain the security guarantees provided by Wasm's sandboxing model. The generated machine code must uphold the same restrictions enforced for the Wasm interpreter, such as ensuring that memory accesses and function calls remain within the bounds of the sandbox. Failure to compile the bytecode in a trusted environment may enable malicious actors to generate machine code that bypasses Wasm's security model, allowing unauthorised access to system resources or execution of malicious code.

4.2.4 Bringing abstraction to operating system services

Wasm does not inherently provide mechanisms for accessing system resources, such as the file system and networking functionalities. As a solution to this shortcoming, Wasm specifications define the concept of *imports*, enabling Wasm applications to execute code not included in the

bytecode by importing functions from the runtime environment. Importing functions bridge the sandboxed Wasm applications and the host system, granting access to system services.

Besides, abstracting OS concerns is a fundamental consideration for Wasm, given that such applications can be executed in both web and standalone contexts, which traditionally had minimal overlap in available system services. Hence, Wasm applications need a way to be deployed across various platforms while the target environment provides a consistent interface for accessing system resources, irrespective of the underlying execution environment.

WebAssembly System Interface (WASI) The Wasm community introduced WASI to address the need for a standardised system interface [97]. WASI defines a set of functions that Wasm applications can import to communicate with OS services through an abstract mediator layer between Wasm applications and the underlying execution environment. This system layer is available to both web-based and standalone Wasm modules, ensuring that applications can access standard OS services, such as file system operations, network communication, and time-related functions, regardless of the execution context. In standalone applications, WASI functions are typically implemented directly by the Wasm runtime. Conversely, in web environments, WASI functions are realised through the browser's Wasm runtime and JavaScript glue code.

Although WASI draws inspiration from the POSIX standard, it is not strictly compatible with it. Developers often bridge this gap by leveraging POSIX-compatible libraries built on top of WASI, such as WASI libc for C/C++ applications [246]. These libraries provide a POSIX-like interface to the underlying WASI functions, abstracting away the differences between the two standards. Alternatively, developers can delegate the mapping between POSIX and WASI to the compiler by targeting a dedicated compilation target, such as the Rust building toolchain [247].

WASI has evolved through several versions since its introduction in 2019. Each version, namely WASI 0, 0.1, and 0.2 (also known as Preview 0, 1, and 2), builds upon the previous version to improve the standardisation of system calls and is detailed below.

- **WASI Preview 0**: WASI 0 [248] laid the foundation for the WASI, acting as an unstable draft that outlined the core types and functions for Wasm applications to access system resources. This initial version was short-lived and not widely referenced in literature, as it was quickly succeeded by WASI 0.1, the first stabilised WASI specification.
- **WASI Preview 1**: WASI 0.1 [249] emerged as the first stable version of WASI and gained widespread support from many web and standalone runtimes. This version introduced 46 functions that comprised various capabilities, including access to process arguments and environment variables, file system interaction, event polling, random number generation, socket interaction, and time retrieval. While WASI 0.1 represented a significant step forward in standardising system calls, it lacked several key functions compared to POSIX, such as the `listen` and `connect` socket functions. These limitations made it difficult to port existing programs to WASI, as runtimes could not fully support server applications without relying on unofficial extensions [250]. Despite these limitations, the work presented in this manuscript leverages WASI 0.1 with such runtime extensions enabled.

- **WASI Preview 2**: WASI 0.2 [251] introduces a redesigned specification that adhered to a *components model*, enabling the composition of mutually distrusted Wasm programs that can be written in different languages [252]. This version includes two distinct worlds: *wasi-cli* and *wasi-http*. The *wasi-cli* world, or the command-line interface world, roughly corresponds to an improved version of WASI 0.1, which is a subset of POSIX functions. On the other hand, the *wasi-http* world is an HTTP proxy world, organised around requests and responses. WASI 0.2 also paves the way for developing additional worlds, such as *wasi-nn* [253], a neural network proposal to include a machine learning API for Wasm.

Capability-based security WASI further enhances the isolation of the Wasm sandbox by using a capability-based security model [100]. In this model, applications are granted specific permissions, known as *capabilities*, to access resources based on the principle of least privilege. Hence, Wasm applications can only access the system resources they explicitly request and are authorised to use by the runtime, minimising the attack surface and preventing unauthorised access to resources. For instance, a Wasm application that needs to read a file must be granted an explicit capability to access that specific file. Similarly, a module requiring network communication must be capable of creating and using a specific socket. WASI ensures that even if a Wasm application is compromised, it cannot access system resources beyond its granted capabilities by enforcing this fine-grained access control policy.

4.3 A building block for interoperable TEE architectures

WebAssembly's platform-agnostic design, strong sandboxing functionalities, and efficient execution modes make it an ideal candidate for enhancing the portability and security of trusted execution environments, as will be further explored in the following sections.

Cross-platform compilation targets Wasm can be compiled from various programming languages using standard toolchains like LLVM. This compilation process generates an intermediate representation, the Wasm bytecode, which abstracts the target execution environment. Wasm enables a higher degree of portability by shifting the responsibility of binding the application to the target environment from the compiler to the runtime. In the context of TEEs, the Wasm runtime abstracts the specific TEE implementation it runs in, enabling interoperability across various CPU architectures. Moreover, Wasm's design is compatible with the trust boundaries of different TEE paradigms, including process-based, VM-based, and partition-based TEEs, as illustrated in Figure 4.2, supporting all the TEE paradigms of mainstream CPU vendors. This compatibility allows Wasm to be a unifying technology for developing trusted applications (TAs) that spans the heterogeneous landscape of the cloud-edge continuum. As depicted in Figure 4.2, TEEs typically require complex and implementation-specific interactions for calling system services. However, WASI presents these interactions through a standardised system layer, enhancing the interoperability of Wasm applications for different TEE implementations.

Strong isolation While TEEs protect the execution environment from the external world, hosting applications within a Wasm sandbox inside a TEE introduces an additional layer of isolation

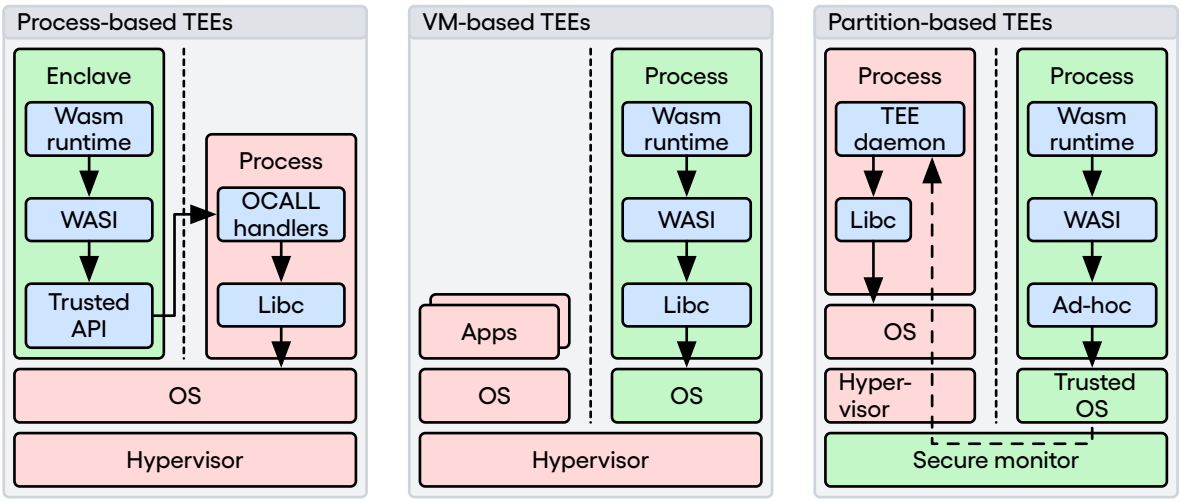


Figure 4.2: The usage of a Wasm runtime within the three primary TEE trust boundaries. The green boxes (▢) are the TEE TCB, the red boxes (▢) are untrusted systems, and the blue boxes (▢) are software components.

that shields other TEE components, such as the trusted OS and co-located TAs, from potential risks posed by malicious or vulnerable code. Wasm enhances the security posture of TEEs by shifting from a unidirectional isolation paradigm to one of mutually distrusting execution environments. In a customer-provider relationship, TEEs ensure the confidentiality of sensitive data and computations from curious providers. At the same time, the Wasm sandbox protects the provider's assets by constraining the customer's software within well-defined boundaries. This two-way isolation promotes a more secure and trustworthy approach to confidential computing.

Legacy support Most traditional applications can be compiled into Wasm thanks to WASI, which eliminates dependencies on the underlying operating system. This compatibility allows applications to be deployed within TEEs without needing TEE-specific SDKs, as the Wasm runtimes handle the mapping of POSIX dependencies to the internal API provided by the TEEs. Besides, this legacy support also promotes the adoption of TEEs by lowering the entry barrier for developers who may not have extensive experience with TEE-specific programming models.

Transparent TEE features Wasm runtimes can replace existing application dependencies with TEE features, hence strengthening some security aspects of the applications. For instance, the runtime can implement file encryption, so interactions with the file system from Wasm applications are transparently secured, providing additional protection for sensitive data. Furthermore, runtimes can expose features only available within TEEs to the hosted applications through standard interfaces, such as a virtual file system for interacting with attestation services. Wasm runtime abstractions, such as WASI, enable developers to leverage the security benefits of TEEs without requiring modifications to their codebase or learning new API, thereby hiding TEE-specific features and integrating them transparently into the execution environment.

Small TCB size Wasm runtimes are generally compact, with some implementations having a binary size under a megabyte, depending on the execution mode for Wasm applications. This

small footprint reduces the likelihood of vulnerabilities within the TEE's trusted computing base (TCB) and makes Wasm well-suited for memory-constrained devices. Furthermore, Wasm runtimes align nicely with the trust boundary of process-based TEEs, which prioritise a smaller TCB compared to coarse-grained VM-based and partition-based TEEs.

Balanced performance and efficiency Wasm runtimes support many execution modes, handling large performance requirements and system constraints. The interpreter mode provides a lightweight and memory-efficient execution environment, well-suited for resource-constrained devices in edge and IoT computing. This mode trades off some performance for reduced resource consumption, enabling Wasm applications to run on systems with limited memory and processing power. On the other hand, more powerful systems can leverage state-of-the-art JIT and AOT compilers to optimise Wasm code for the target platform. Such tools analyse the Wasm bytecode and generate highly efficient machine code tailored to the specific hardware architecture, bringing Wasm applications performance close to native applications.

4.4 A rich ecosystem

Wasm runtimes exist for web applications and standalone execution, targeting different computing environments. In the web context, browsers typically extend their JavaScript engines to support Wasm, such as V8 for Chromium-based browsers and SpiderMonkey for Firefox [254, 255]. Conversely, standalone execution relies on dedicated runtimes specifically designed for Wasm. While prior work has explored executing Wasm applications using V8 within Intel SGX enclaves, this approach relied on a library OS for interoperability with the TEE, resulting in a large TCB [120]. Instead, this dissertation focuses on purely standalone runtimes for embedding Wasm within TEEs, considering the disk and memory size constraints necessary for implementing a minimal TCB and excluding unnecessary components like a JavaScript engine.

4.4.1 An overview of the standalone runtimes landscape

The following paragraphs explore various publicly available standalone runtimes and compare some of their characteristics to identify the most suitable candidates for integration with TEEs.

Wasm3 A highly lightweight and high-performance Wasm interpreter designed for resource-constrained environments such as embedded systems, IoT devices, and other limited-resource scenarios [118]. As a micro-interpreter optimised for size and fast startup latency, Wasm3 can execute Wasm applications in restricted memory environments, making it suitable for constrained edge devices like Arduino. Its small set of dependencies and small codebase enable integration within process-based TEEs like Intel SGX enclaves. Wasm3 is maintained by the open-source community and can be used as a library for various programming languages.

WAVM A standalone WebAssembly runtime written in C++, designed to execute Wasm applications on x86-64 architectures [256]. WAVM can use JIT and AOT compilers, leveraging the LLVM infrastructure to generate performant native code. The tight coupling of the JIT compiler

with the LLVM infrastructure introduces challenges when embedding it inside some TEEs such as Intel SGX and Arm TrustZone because of the many dependencies of LLVM. WAVM's public GitHub repository has remained stalled since 2022, indicating a lack of active development and community support compared to the other runtimes analysed in this section.

Wasmer A fast Wasm runtime implemented in Rust, designed for lightweight and portable Wasm containers to run from desktop to the cloud and edge computing [257]. It offers a flexible architecture that supports multiple compilation strategies, including JIT and AOT compilers, which adapt to different environments, from desktop applications to cloud, edge, and IoT devices. Wasmer integrates with several backend compilers, such as LLVM, Cranelift [258], and Singlepass [259], each offering distinct advantages in compilation speed and generated machine code quality. The Singlepass backend provides exceptionally fast compilation times, making it suitable for scenarios that prioritise quick startup, such as blockchain applications. Cranelift, the default backend, balances compilation speed and performance, delivering good execution speeds with reasonable compilation times. For applications demanding the highest possible performance, the LLVM backend offers state-of-the-art optimisation features, albeit with slower compilation times. Moreover, through its high-level language API, Wasmer can be used as a standalone runtime or embedded as a library in various programming languages.

WasmEdge A lightweight, high-performance and extensible Wasm runtime written in C++ and designed for cloud-native, edge, and decentralised applications [260]. It primarily aims to power serverless apps, embedded functions, microservices, smart contracts, and IoT devices. WasmEdge uses LLVM-based AOT and JIT compilers for executing Wasm applications, offering near-native speed. As WasmEdge focuses on cloud-native and edge computing applications, one of its major use cases is to start a VM instance from a host application, hence providing an SDK for various programming languages to start and invoke WasmEdge functions. This runtime also comprises advanced functionalities for supporting container orchestration tooling like Kubernetes.

Wasmtime A standalone Wasm runtime developed in Rust by the Bytecode Alliance is designed for executing Wasm code outside the web environment [261]. Similarly to Wasmer, it leverages Cranelift, a JIT compiler with similarities to LLVM, for efficient code generation. Lucet [262], a former Wasm runtime also developed by the Bytecode Alliance, was merged in 2020 into Wasmtime, bringing AOT compilation for Wasm applications. Wasmtime can be configured to provide more fine-grained control over resources like CPU and memory consumption, supporting tiny environments as well as massive servers with many concurrent instances. Besides, Wasmtime can be used as a command-line utility or embedded as a library in various programming languages through the provided wrappers.

WAMR The WebAssembly Micro Runtime (WAMR) is a standalone Wasm runtime by the Bytecode Alliance [117]. WAMR offers all the execution modes mentioned in Section 4.2.3, which are two interpreted execution modes, one optimised for speed and the other for memory efficiency, as well as two binary execution modes: JIT and AOT compilation, both leveraging

the LLVM compiler infrastructure. Implemented in C with a small footprint and minimal external dependencies, WAMR is well-suited for resource-constrained embedded execution environments, such as IoT devices and TEEs. WAMR also targets Wasm execution on server-grade machines with its near-native speeds thanks to JIT and AOT modes. Its runtime binary size is approximately 60 KiB for the interpreter modes and 30 KiB for AOT mode. Notably, WAMR can be linked with Intel SGX enclaves, making it an attractive choice for deploying Wasm applications in TEEs. Finally, this runtime supports many architectures and platforms, including x86, Arm, RISC-V, and many OS, such as Windows, Linux, macOS, and real-time OS like Zephyr.

4.4.2 A summary of runtimes characteristics

Table 4.1 compares some key characteristics of the Wasm runtimes previously analysed in this chapter, highlighting their suitability for embedding into the state-of-the-art industrial TEEs. Each feature can either be missing (○), partially (◑), or fully (●) available.

Features	Wasm3	WAVM	Wasmer	WasmEdge	Wasmtime	WAMR
Interpreter mode	●	○	○	●	○	●
just-in-time (JIT) mode	○	●	●	●	●	●
ahead-of-time (AOT) mode	○	●	●	●	●	●
WASI support	●	●	●	●	●	●
CPU support for x86	●	●	●	●	●	●
CPU support for Arm	●	○	●	●	●	●
CPU support for RISC-V	●	○	●	●	●	●
Runtime still maintained	●	○	●	●	●	●
TEE support	○	○	○	○	○	◑⋆
Programming language	C	C++	Rust	C++	Rust	C
Time in existence	5 years	9 years	6 years	5 years	8 years	6 years
GitHub # commits	1.7k	2.4k	17.8k	3.6k	13.5k	2k
GitHub # stars	7.2k	2.6k	18.4k	8.3k	15k	4.7k

⋆ WAMR provided basic support for Intel SGX, which did not include WASI for SGX when starting this dissertation.

Table 4.1: The comparison of the state-of-the-art Wasm standalone runtimes, as of August 2024.

4.4.3 A balance of performance and lightweightness for TEEs

The decisive factors for selecting a runtime dedicated to TEEs are: (i) the programming language used for its development, as many TEEs limit their SDK to C or C++, such as Intel SGX and OP-TEE for Arm TrustZone; (ii) the runtime's minimal external dependencies, including system calls and libraries required for compilation; (iii) support for a near-native execution mode like AOT, which eliminates the need for embedding a JIT compiler and, therefore, min-

imises the TCB; (iv) a small size, which allows the runtime to fit within the memory constraints typically imposed by some TEE implementations; and (v) existing TEE support.

Based on these criteria, WAMR emerged as the best choice. When starting this dissertation, WAMR had basic built-in support for Intel SGX enclaves, easing the integration of Wasm and SGX. However, WAMR lacked WASI support for SGX, and implementing such functionalities in a privacy-preserving manner is crucial, as programs need to communicate outside the TEEs to interoperate with other software components and persist confidential information. Consequently, enabling WASI support for TEE implementations is a primary objective of this thesis.

Wasm is ideal for envisioning the cloud-edge continuum as an interoperable, scalable, and distributed system where software can be deployed on any peer, regardless of the underlying platform. This technology has the potential to transform the development lifecycle of future applications, enabling developers to focus on delivering business value instead of dealing with each distinct TEE infrastructure, which has rapidly evolved over the past decade with the advent of confidential computing. Thanks to its lightweight specifications, large support of programming languages, various execution modes offering near-native speed, strong sandbox, and versatile abstraction of the underlying system and ISA, it becomes clear that Wasm and trusted computing can be the bedrock for software development in large-scale systems in the coming years.

We noticed that prior research has only superficially measured Wasm's performance and scalability, primarily due to the novelty of this technology. The next part of this manuscript provides an in-depth analysis of Wasm applications deployed on the web and in standalone runtimes to determine whether this technology can compete with native execution. Furthermore, the following part lays the foundations of creating trusted runtimes in several TEE implementations, highlighting the benefits and constraints of using Wasm for confidential computing.

Part II

WebAssembly as an efficient and attestable abstraction

Chapter 5

A use-case of WebAssembly: IncaMail

This chapter examines the performance and application of WebAssembly (Wasm) for cryptographic operations in web browsers compared to JavaScript implementations. To demonstrate the benefits of Wasm, various cryptographic operations of IncaMail, a secure email service by Swiss Post, are offloaded from the server side to the client's browser, yielding several advantages, such as reduced computational load, relaxed trust assumptions, and per-message encryption key policies. The results presented in this chapter highlight the substantial performance improvements that Wasm applications can deliver to web-based services.

Chapter outline

5.1 Introduction . . . 72
5.2 Background . . . 72
5.3 Use-case: IncaMail . . . 73
5.3.1 Offloading cryptographic operations . . . 74
5.3.2 Proposed workflow . . . 75
5.4 Evaluation . . . 77
5.4.1 Experimental setup and methodology . . . 77
5.4.2 Micro-benchmarks: encryption scheme . . . 77
5.4.3 Scalability of the encryption cipher . . . 78
5.5 Synthesis and next steps . . . 79

5.1 Introduction

The widespread adoption of Wasm in modern web browsers provides an execution environment with near-native performance for web applications. However, as an emerging technology, the performance of Wasm for real-world applications remains largely unexplored in academic literature. We take the opportunity of collaborating with the Swiss Post to explore further and measure how Wasm enhances the performance and the security model of one of their web applications by compiling OpenSSL, a reputable cryptographic library, in Wasm so it can be executed directly in the end-users' device. This browser-based Wasm implementation serves as the starting point for the ongoing research of this dissertation into integrating security features in standalone Wasm runtimes and trusted execution environments (TEEs).

Secure messaging has become increasingly important in today's digital landscape, where privacy and security are paramount concerns. Encryption schemes are critical in ensuring the confidentiality and integrity of messages transmitted over untrusted networks. However, traditional email encryption approaches suffer from various vulnerabilities. Efail attacks [263] exploit the lack of message authentication in encrypted emails to manipulate ciphertexts and exfiltrate plaintext data through maliciously crafted HTML backchannels. To counter such threats and enhance the security of email communication, Swiss Post introduced IncaMail [145], a secure email service designed for transmitting legally binding, encrypted, and verifiable emails. IncaMail uses an authenticated encryption with associated data (AEAD) encryption scheme to guarantee message privacy and authentication between servers. The current architecture offers a robust solution against Efail attacks. However, it also presents limitations due to its centralised nature, such as high computational load on the server, sustained network traffic between clients and servers, and the need for a fully trusted server that handles plaintext messages.

This chapter presents an experimental IncaMail architecture revision that offloads most cryptographic operations to clients' browsers using Wasm. Our proof-of-concept prototype fully embeds the industry-standard OpenSSL cryptographic library, compiled in Wasm and embedded in the web browser. Our experimental evaluation shows significant performance improvements over alternative approaches based on JavaScript, while preserving linear time complexity with respect to message and file attachment size. Client-side encryption offers numerous performance and architectural benefits, including reduced computational load and, therefore, lower energy footprint, relaxed trust assumptions, and per-message encryption key policies. This work contributes to the advancement of secure email communication services and highlights the potential of Wasm to enhance the performance and security of such systems.

5.2 Background

Efail attacks [263], a security vulnerability disclosed in 2018, compromises the confidentiality of email encryption protocols, specifically OpenPGP and S/MIME, by exploiting implementation flaws and allowing adversaries to decrypt messages. They exploit weaknesses in the interaction between email clients and encryption plugins, as well as the properties of the encrypted messages themselves. The attack primarily targets HTML-rendering email clients that use OpenPGP

or S/MIME encryption. Two attack variants exist: the *direct exfiltration* attack and the *CBC/CFB gadget* attack. Both variants leverage the concept of malleability in encrypted messages, allowing attackers to manipulate ciphertext without knowing the corresponding plaintext. In the direct exfiltration attack, the adversary alters the encrypted email by injecting an image tag with a crafted URL containing decrypted content once the email client processes the modified email. When the victim's email client decrypts the message and loads the injected image, the plaintext message is inadvertently sent to the attacker-controlled server. The CBC/CFB gadget attack is more sophisticated and relies on manipulating the block cipher modes of operation in OpenPGP or S/MIME. The attacker induces specific plaintext patterns by carefully crafting the ciphertext and exploiting the lack of integrity protection, which, combined with HTML tags, can be used to exfiltrate the decrypted content similarly to the direct exfiltration attack.

The risk posed by the Efail attacks can be mitigated with countermeasures at the email client and protocol levels. Such measures include: (i) disabling HTML rendering: since the Efail attacks rely on the HTML rendering features of email clients to exfiltrate decrypted data, disabling HTML rendering in the client settings can effectively prevent the attacks; (ii) integrity protection: implementing cryptographic mechanisms, such as authenticated encryption with associated data (AEAD) or message authentication code (MAC), can ensure the integrity of encrypted messages and thwart attempts to manipulate ciphertexts; and (iii) secure implementations: email client developers and encryption plugin providers must adhere to the best practices for implementing encryption protocols, such as properly handling decryption errors and avoiding the leakage of plaintext data through external resources. The Efail attacks highlight the importance of correct implementation and usage of encryption protocols in email communications. This chapter demonstrates how authenticated encryption at the client-side browser using Wasm effectively addresses the risks associated with Efail while ensuring the ongoing confidentiality of encrypted email communications.

Prior work proposed generic mitigations against the two variants of Efail attacks [263–266]. Typical mitigations, as presented by Schwenk *et al*. [266], involve checking the context during decryption, such as the Simple Mail Transfer Protocol (SMTP) headers and Multipurpose Internet Mail Extensions (MIME) structure. Their solution is implemented into the email client, for example, Thunderbird, and uses the AEAD encryption scheme. The AEAD's authenticated encryption ensures the confidentiality and integrity of the emails, preventing the malleability gadget attack. Additionally, it leverages the AEAD's associated data to protect against the direct exfiltration attack. Building upon their work, we design and implement a solution that is not limited to a specific email client. Instead, it is designed for web-based senders and receivers, potentially accessible via mobile devices, as Wasm is executed in a standard browser. Moreover, we aim to provide additional support through an Outlook add-in as an extension of our approach, as Microsoft supports Blazor WebAssembly for Office add-ins [267].

5.3 Use-case: IncaMail

The Swiss Post developed IncaMail, a service offering a secure way of transmitting information via emails. The primary advantage of this communication channel is that it offers authenticated

encryption, which mitigates Efail attacks. Users can send messages using three input channels: (i) a web interface; (ii) an Outlook add-in; and (iii) a dedicated web API.

Current workflow When a customer of IncaMail sends a secure email, the secure message is transmitted to IncaMail's backend servers for encryption using a symmetric cipher, such as Advanced Encryption Standard (AES). The message is encrypted for each recipient. The ciphertexts are embedded within standard emails as attachments and subsequently sent individually to the respective recipients. The encryption keys are retained on IncaMail's premises, while the secure messages are not stored within the Swiss Post infrastructure, except in some instances for caching purposes. Upon reception of the email, one or more recipients open the attachment. This attachment is an HTML file containing an HTML form with predefined hidden fields, which include the ciphertext, a MAC, and other associated data. A given recipient submits the form to the IncaMail server, which internally retrieves the encryption key, decrypts the submitted information, and displays the plaintext of the secure message. Additionally, secure messages may also include secure file attachments, which are processed with a similar encryption mechanism.

The centralised architecture has several limitations. Firstly, all cryptographic operations occur solely on the server, demanding additional computing resources from Swiss Post compared to performing the operations on the client's device. Secondly, this approach generates more network traffic than decrypting messages on the client, as all messages and corresponding ciphertexts are transferred to and from the Swiss Post infrastructure. Thirdly, the IncaMail server must be a fully trusted entity since it has access to all messages in plaintext, allowing Swiss Post to potentially read and manipulate messages in case of errors or compromise. Finally, group messages are encrypted multiple times, once per recipient, because all messages are encrypted with a private key bound to each recipient.

5.3.1 Offloading cryptographic operations

In this work, we evaluate a revised architecture that offloads certain cryptographic operations to the customers' browsers. This approach offers several advantages over the current architecture. Notably, it reduces the CPU and volatile memory usage, as the server no longer needs to encrypt and decrypt the secure message. Additionally, it reduces the trusted computing base (TCB) since only the endpoint that receives and delivers encryption keys must be trusted, given that plaintexts are no longer sent to the IncaMail servers. Furthermore, we improved the encryption scheme of IncaMail to share a common encryption key for all the recipients of a given message, which reduces the amount of data to store in the IncaMail infrastructure.

Although the W3C's web cryptography API [268] facilitates access to various cryptographic operations in modern browsers, we opted for Wasm as a comprehensive solution for managing cryptographic computations, using OpenSSL [269] as the cryptography library, due to a number of considerations. Firstly, Swiss Post maintains numerous technologies for servicing IncaMail, such as a web interface and an Outlook add-in. Wasm facilitates the use of well-established cryptography libraries across various platforms and browsers, ensuring compatibility even when the native web cryptography API may not support specific algorithms. As OpenSSL maintains a consistent API across platforms, developers can leverage existing code

and knowledge when implementing cryptographic operations. This consistency eases development and maintenance compared to using different API for various platforms. Secondly, Wasm is designed for efficiency, portability, and speed. While it may not match the performance of native code, it significantly outperforms JavaScript in terms of execution speed. This chapter demonstrates that using OpenSSL yields superior performance compared to JavaScript-based alternatives. Lastly, OpenSSL offers greater control and flexibility than the web cryptography API, as it grants developers access to a broader range of cryptographic primitives and fine-tunes their implementations to satisfy specific requirements. This advantage frees Swiss Post from being constrained by browser implementation choices.

Compiling OpenSSL for Wasm was not trivial, as the official OpenSSL distribution lacked support for Wasm compilation. To address this issue, we used Emscripten [244], a toolchain for compiling C/C++ code to Wasm. However, certain features of OpenSSL required modifications to ensure compatibility with the Wasm environment. Specifically, we disabled hardware acceleration, as WebAssembly System Interface (WASI) does not provide direct access to assembly instructions. New WASI worlds like wasi-crypto [270] aim to provide a direct link with the runtime, enabling Wasm to offload cryptographic operations to hardware. Furthermore, we opted to disable multithreading, as it was irrelevant to our specific requirements.

We adapted the IncaMail architecture by implementing several changes. The key generation operations are now delegated to the client side, while a new server-side RESTful API is responsible for managing the keys. This API, named the *cryptographic API*, acts as the TCB of IncaMail. The encryption and decryption of secure messages and file attachments are also offloaded to the client side, ensuring that the plaintext of messages no longer passes through the IncaMail infrastructure. Consequently, we have improved the threat model of the IncaMail backend infrastructure, allowing it to adhere to an honest-but-curious model in which the system can inspect exchanged information without compromising the confidentiality of messages.

5.3.2 Proposed workflow

We developed a proof-of-concept prototype of the proposed architecture and present the new workflows for using IncaMail with client-side encryption and Wasm. Figure 5.1 depicts the secure message-sending workflow. The server-side software consists of the existing *IncaMail API* and the new cryptographic API. The client-side software includes three layers for facilitating integration with various platforms: (i) the functionality layer, which contains generic event handlers invoked by the integration layer; (ii) the kernel layer, which provides the essential cryptographic functionality for message encryption and decryption; and (iii) the integration layer, responsible for linking the graphical user interface to the functionality layer.

The sending encryption process involves the following steps: the user authenticates, composes a secure message, optionally appends secure files, selects recipients, and submits the form (①); the kernel (*i.e.*, OpenSSL compiled in Wasm) generates an encryption key and encrypts the file attachments and the message (②–③); the encryption key is transmitted to the cryptographic API (④); the ciphertexts are forwarded to the IncaMail API, which embeds them in an HTML file for assisted decryption (⑤); and finally, the IncaMail API sends a standard email to the recipient, notifying them that the secure message is awaiting on the platform (⑥).

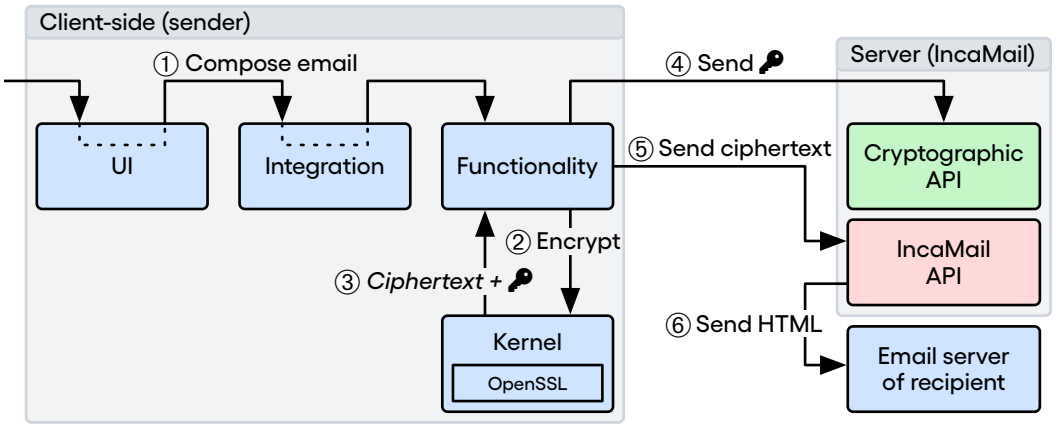


Figure 5.1: The workflow of sending a secure message using IncaMail. The green boxes (▢) are the TCB of IncaMail, the red boxes (▢) are untrusted, and the blue boxes (▢) are components outside IncaMail servers.

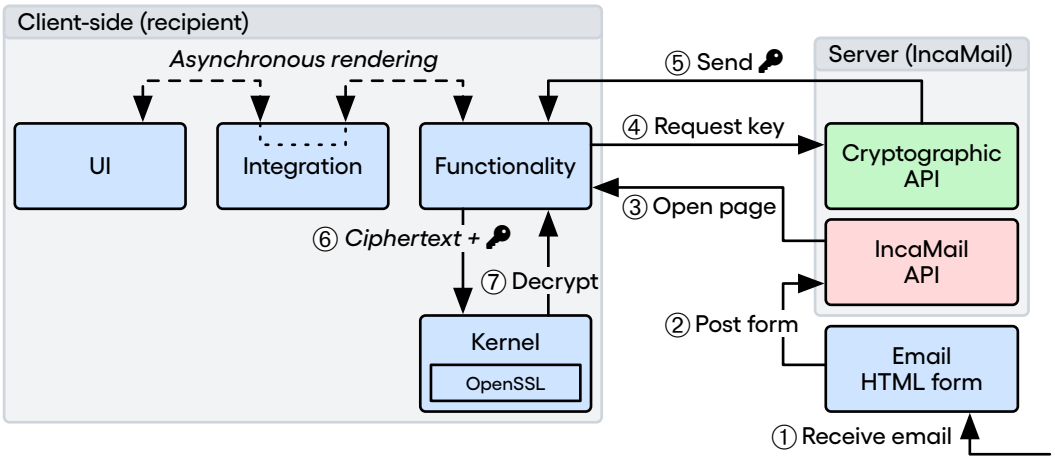


Figure 5.2: The workflow of receiving a secure message using IncaMail. The green boxes (▢) are the TCB of IncaMail, the red boxes (▢) are untrusted, and the blue boxes (▢) are components outside IncaMail servers.

Figure 5.2 illustrates the process of a recipient reading a secure message, which is described by the following steps: the recipient receives an email containing an HTML attachment (①); upon opening the HTML file, the recipient is prompted to read the secure message by clicking a button, which submits the ciphertext and associated data to the IncaMail API (②); the IncaMail API then serves a page that loads the client-side software (③); the client-side software requests the encryption key (④); the cryptographic API of IncaMail sends the decryption key (⑤); the kernel decrypts the message and file attachments using that key (⑥); and they are rendered asynchronously (⑦). As the decryption is handled asynchronously on the end-user device, the page displays the message and remains unblocked while the file attachments are decrypted. This enhances the responsiveness of the rendering compared to the server-side approach.

5.4 Evaluation

In this section, we present an evaluation of our proposed solution, aiming to address the following research questions: (i) How does the performance of the encryption scheme implemented in Wasm compare to plain JavaScript? (ii) Does the time complexity of the encryption cipher remain linear when using Wasm? To answer these questions, we employ a micro-benchmark that measures the encryption and decryption times for payloads of varying sizes (§5.4.2). Besides, we assess whether the encryption scheme maintains linear time complexity (§5.4.3).

5.4.1 Experimental setup and methodology

We benchmarked different mobile phones, including Apple iPhone Pro 12/13/14 (running iOS 16) and Google Pixel 5/Pro 6/Pro 7 (with Android 12 for the former and Android 13 for the latter two). We leveraged the online platform LambdaTest [271] to execute the benchmarks on actual devices. As the benchmarks solely operate on the client side, we did not consider network latency. We note that Safari uses the WebKit [272] engine, whereas Chrome uses Blink [273], a fork of WebKit. The use of different browser engines is imposed by Apple policy, forcing iOS devices to use WebKit strictly [274]. We compiled OpenSSL v3.0.5 using Emscripten v3.1.34.

5.4.2 Micro-benchmarks: encryption scheme

We created a micro-benchmark to assess the performance of our Wasm implementation. We compare against a pure JavaScript implementation by Stanford [275], which explored various optimisation strategies for executing cryptographic operations within a JavaScript engine. The benchmark uses AES cipher with a key length of 128 bit for encrypting and decrypting payloads generated by a pseudorandom number generator (PRNG). Payload sizes ranged from 1 MiB to 20 MiB, with the maximum corresponding to IncaMail's file size limit for email attachments. We measured the time taken by the cryptographic library to compute ciphertext and plaintext using the function `performance.now` for the JavaScript implementation, and the function `clock_time_get` with a monotonic clock for Wasm thanks to WASI. The benchmark disregarded the setup and teardown time for individual runtimes. Results were aggregated by device type, specifically *iPhone* and *Pixel*, due to the negligible differences observed among the models. Consequently, we focused on the results obtained from the iPhone Pro 14 and Google Pixel Pro 7 devices for this analysis. Each experiment was executed ten times per device, with the mean value used to determine the average outcome.

Figures 5.3a and 5.3b present the results obtained using iPhone and Pixel devices, respectively. We first observe that the speedup remains consistent, irrespective of the payload size for the encryption or decryption. On an iPhone, the encryption speedup using Wasm compared to JavaScript is 13.9×, while on a Pixel device, it is 6.9×. Although we did not explore the precise cause of superior performance on Apple devices, it is suspected to be due to a more optimised Wasm runtime for executing OpenSSL. The absolute time taken by an iPhone to encrypt 20 MiB of data using Wasm is 307.2 ms, whereas on a Pixel device, it is 786.9 ms, resulting in a ratio of 2.6 between the two systems. In contrast, when comparing the same operation using pure

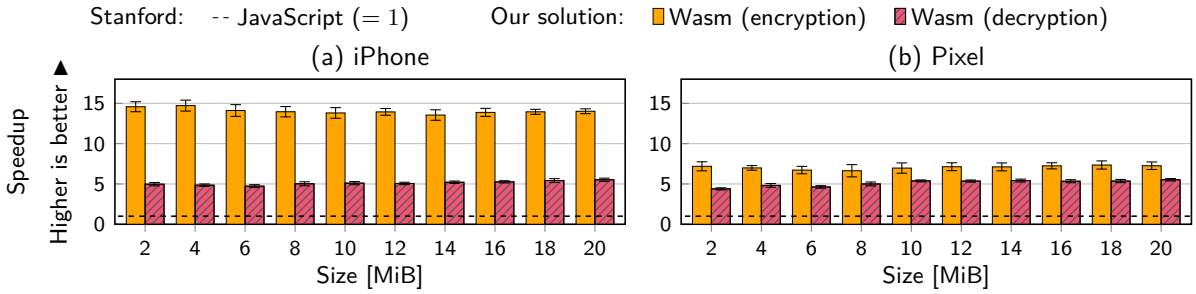


Figure 5.3: The speedup of the Wasm encryption and decryption operations, compared to a pure JavaScript implementation.

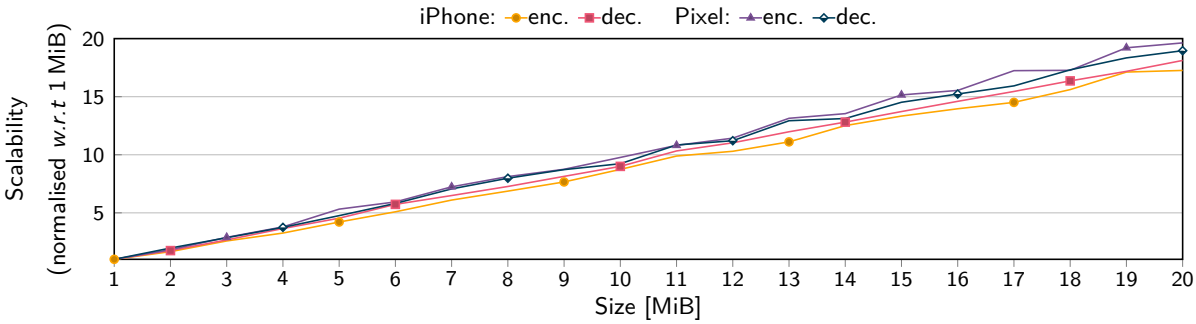


Figure 5.4: The scalability of the encryption cipher using Wasm, relative to the plaintext.

JavaScript, we observe a ratio of 1.3 (4305.6 ms on an iPhone and 5700.4 ms on a Pixel device). The decryption speedup using Wasm is 5.1× for both iPhone and Pixel devices. These findings illustrate that Wasm significantly outperforms plain JavaScript code in cryptographic operations, promoting an efficient offloading of server-side operations to end-users' device.

5.4.3 Scalability of the encryption cipher

The deployment of emerging technologies in constrained environments like web browsers may incur additional overheads and performance penalties. We further analysed the results of our benchmarks to assess the scalability of cryptographic operations, specifically examining whether the time complexity of the implementation increases with larger payload sizes. In this experiment, we selected the execution time to encrypt and decrypt payloads ranging from 1 MiB to 20 MiB, subsequently dividing the obtained measurements by the time taken to encrypt 1 MiB.

Figure 5.4 depicts the scalability of the previously-conducted benchmark for encryption and decryption operations by device type. The results reveal that the time complexity of these cryptographic operations is linear when using Wasm, *i.e.*, the complexity can be expressed as $\mathcal{O}(n)$, where n denotes the number of bytes to process. Given that the execution time of AES algorithm is linear, the strategy of offloading these cryptographic operations to the client side does not compromise the user experience, as the implementation remains efficient and performant.

5.5 Synthesis and next steps

This chapter presents a proof-of-concept that demonstrates the offloading of cryptographic operations from IncaMail's centralised architecture to clients' browsers. By compiling OpenSSL in Wasm, we achieved a significant speedup in encryption and decryption tasks compared to JavaScript implementations, while maintaining a linear time complexity relative to secure message and file attachment sizes. Furthermore, the implementation shares a common encryption key for all recipients of a given message, resulting in non-volatile memory savings. The new architecture reduces Swiss Post's resource usage, minimises client-server traffic, and strengthens the security posture by reducing the TCB. The findings in this chapter indicate that Wasm is a viable and efficient solution for offloading cryptographic operations to the client side, improving the performance and security of secure email communication services such as IncaMail.

The successful compilation and usage of OpenSSL in Wasm with nearly no change to the codebase demonstrates the feasibility of porting complex libraries to this platform. OpenSSL's interaction with system services, such as the PRNG, is done through WASI, which is supported by web and standalone Wasm runtimes. This work gives confidence in the viability of adapting complex libraries like OpenSSL to more constrained environments, such as TEEs.

The upcoming chapter shifts focus to evaluating the performance of Wasm within standalone runtimes. It explores the integration of Intel SGX, a process-based trusted execution environment that provides hardware-based security guarantees to the runtime by allowing the design of a minimal TCB within a trusted environment. The chapter also considers Wasm dependencies such as WASI and the attestation service. Standalone runtimes can be optimised to consume less than one megabyte of disk space, making them well-suited for integration with SGX enclaves.

Chapter 6

A trusted WebAssembly runtime for Intel SGX: Twine

WebAssembly (Wasm) has extended web development, allowing the creation of web components using system programming languages and facilitating the porting of existing libraries and applications to browsers. Wasm's presence also extend beyond the web, with standalone runtimes executing Wasm-compiled programs as traditional applications. These runtimes offer an ideal environment for confidential computing, thanks to Wasm's near-native performance, portability, capability-based security model, and strong sandboxing, while presenting a minimal trusted computing base (TCB). This chapter introduces Twine, a trusted runtime that executes Wasm applications using Intel SGX, a process-based trusted execution environment (TEE) providing strong confidentiality and integrity guarantees along with an attestation mechanism.

Chapter outline

6.1 Introduction . . . 82
6.2 Related work for Twine . . . 83
6.3 Twine design . . . 85
6.3.1 Requirements elicitation . . . 85
6.3.2 Threat model . . . 85
6.3.3 Architectural overview . . . 86
6.4 Implementation details . . . 88
6.4.1 Memory allocation . . . 89
6.4.2 Communication support . . . 90
6.4.3 File management support . . . 91
6.4.4 Attestation . . . 92
6.5 Demonstrating Twine with a distributed financial technology . . . 93
6.6 Evaluation . . . 95
6.6.1 Experimental setup . . . 95
6.6.2 Micro-benchmarks: PolyBench/C . . . 96
6.6.3 Micro-benchmarks: network stack . . . 98
6.6.4 Macro-benchmarks: SQLite . . . 100
6.6.5 Macro-benchmarks: Twine for credit scoring . . . 107
6.7 Security analysis . . . 108
6.8 Synthesis and next steps . . . 110

6.1 Introduction

Data confidentiality and secure code execution are fundamental components for many organisations in the current data-driven and interconnected landscape. TEEs like Intel SGX/TDX [102, 107], Arm TrustZone/CCA [104, 108], AMD SEV/-ES/-SNP [182, 184, 185] and TEEs for RISC-V [103, 105, 196, 200] gathered much attention lately as they provide hardware support for secure code execution within special hardware constructs, *i.e.*, trusted applications (TAs), shielded from the outside world, including malicious or compromised operating systems, hypervisors and privileged users. In a typical TEE usage model, multiple entities do not always trust each other: (i) the *TEE service provider* that provides the TA for confidential processing; (ii) the *infrastructure provider* hosting the TEE-based service; and (iii) the *data owner* who outsourced its data for remote processing. TEEs guarantee security for service providers and data owners against malicious infrastructure providers. However, there are no consolidated approaches to ensure trust to data owners or infrastructure providers that completely mistrust the TEE service provider. This is even more problematic when the TA is private and cannot be exposed. Programs may have exploitable bugs or write data out of the TEE through corrupted or dangling pointers.

To cope with the situation explained, researchers explored well-studied techniques like software fault isolation (SFI). SFI [276] establishes a logical protection domain by inserting dynamic checks before memory and control-transfer instructions, which are then verified at runtime. However, SFI requires either source code access, binary rewrite, or complex manipulation of the process memory during execution, which makes this technique usually unsuited for production use [277]. In contrast, WebAssembly (Wasm), initially conceived for executing high-performance native code in browsers, allows building a memory-safe, lightweight, and portable sandboxed execution environment based on restricted memory access and control flow with limited usage of resources. Ryoan [278] and Deflection [279] are notable solutions which combined SFI with TEEs. However, both suffer from performance and usability issues as these runtimes require to modify the source code to operate. Regarding Wasm within TEEs, we highlight AccTEE [64] and SGX-LKL [120], as two solutions that execute Wasm-compiled applications inside secure enclaves. AccTEE provides resource accounting for Wasm bytecode but lacks comprehensive OS-level sandboxing, relying instead on SGX-LKL for enclave execution. SGX-LKL serves as a library OS designed for generic code execution but does not offer sandboxing functionalities and exposes a large TCB due to its integration with the Linux kernel library.

This chapter presents TWINE, a runtime that supports the execution of Wasm-compiled applications in trusted execution environments, including potentially legacy software. It provides a two-way sandbox, ensuring security for both the service provider and the infrastructure owner. TWINE comes with an extended WebAssembly System Interface (WASI), which allows the sandboxed applications to issue filtered and controlled OS services. We currently support Intel SGX as a TEE: TWINE translates at runtime WASI operations into equivalent native OS calls or functions in secure libraries specifically built for Intel SGX. In particular, TWINE maps all file operations to Intel Protected File System (IPFS) [280], and persisted data is transparently encrypted and never accessible in plaintext from outside an enclave, thus shielding against data exfiltration. Furthermore, our solution provides configurable and secure communication functionalities to the Wasm binary running inside the TEE using raw TLS or HTTPS. Last but not least, TWINE

includes attestation features, which allow us to verify the integrity of the entire runtime and, most importantly, how OS services are offered to the Wasm module. While a TEE provides a secure hardware execution runtime in the processor, TWINE provides a secure software runtime (sandbox) nested within the TEE, using WASI for interoperability with regular API and abstracting the underlying environment from the application.

TWINE is evaluated with several micro- and macro-benchmarks. The performances of TWINE are compared against existing alternatives, with and without secure operations inside a TEE. Our results reveal that TWINE performs on par with systems also exploiting TEEs and providing similar security guarantees while offering programmers greater freedom. We observed performance overheads for some specific workloads due to execution in the TEE. Besides, we integrated and deployed in production the resulting trustworthy runtime in the context of a fintech company, which, in short, computes real-time credit scoring, achieving 0.7–1.17× the performance of their original native software. We believe this penalty is largely compensated by the additional security guarantees and the interoperability granted by our trusted WASI layer.

The contributions presented in this chapter are summarised as follows: (i) the first fully open-source implementation of a general-purpose Wasm runtime environment within Intel SGX enclaves with support for encrypted file system and networking, as well as built-in primitives for remote attestation, (ii) an extensive experimental evaluation, shedding light on performance costs and associated bottlenecks, as well as a real-world integration in an industry-battled scenario, (iii) a proposal for improving IPFS and a showcase of the derived performance improvements, and (iv) a fully upstreamed solution to the original Wasm runtime (*i.e.*, WAMR), which is open-source, maintained, and ready for use in production environments.

6.2 Related work for Twine

Building upon the background part presented earlier in this thesis (§I), we survey more specialised related work for TWINE, focusing one three key criteria. First, we look at systems with dedicated support for Wasm in TEEs. Then, we review proposals for generic TEE support for language runtimes. Finally, we investigate alternative sandboxing solutions embedded in TEEs.

Wasm and TEEs ACCTEE [120] runs Wasm inside Intel SGX, with the specific goal of implementing trustworthy resource accounting under malicious OSes. It leverages the SGX-LKL [64] library OS and V8 JavaScript/WebAssembly engine to execute Wasm binaries inside SGX enclaves. Their two-way sandbox (firstly from disjoint memory spaces for Wasm modules, and secondly from SGX itself) is similar to TWINE's double-sandboxing approach (explained in Section 6.4). ACCTEE lacks a WASI layer and instead uses custom JavaScript for I/O interfacing, managed by SGX-LKL. This absence of a WASI layer presents two issues: firstly, a performance slowdown due to the constant call of JavaScript for external interactions, and secondly, a sandboxing challenge since ACCTEE uses Emscripten for Wasm compilation, which can invoke system calls. In contrast, TWINE includes WASI to securely handle data persistence and communication.

Se-Lambda [281] is a library built on top of OpenLambda [282] to deploy Wasm-based serverless programs over function as a service (FaaS) platforms. It shields the FaaS gateway and the code of the deployed functions in enclaves, providing anti-tampering and integrity guarantees. Furthermore, it protects against attacks from a privileged monitoring module that intercepts and checks system call return values. Similar defence mechanisms can be integrated into TWINE.

Enarx [283] represents an open-source TEE runtime that eases the execution of Wasm binaries across a few TEEs, including those compatible with Intel and AMD processors. Although both Enarx and TWINE extend support for network and attestation functionalities, there are notable distinctions between the two solutions. Specifically, Enarx lacks file system support and solely relies on a just-in-time (JIT) compiler. In comparison, TWINE provides a robust safeguard for data at rest, as well as an ahead-of-time (AOT) compiler that ensures fast start-up and execution.

Veracruz [284] protects workloads running on Arm CCA and AWS Nitro [285]. Unlike TWINE, Veracruz is a framework for developing Wasm-compiled applications in a confidential cloud computing context rather than an arbitrary execution environment for unmodified software.

Embedding language runtimes in TEEs There have been many efforts to embed other language runtimes into TEEs like .NET [79], Python [286], Java [80], JavaScript [287, 288], and Lua [288, 289]. TWINE deploys a lightweight and versatile Wasm runtime inside an SGX enclave, able to execute AOT-compiled Wasm applications for optimal performance. Additionally, we developed a WASI layer to enable any WASI-compliant application to run inside our runtime.

Sandboxing inside TEEs Lastly, we report prior work that applied data confinement solutions based on SFI inside TEEs to protect against untrusted enclave service providers. Ryoan [278] introduced a distributed sandbox by adapting the Google Native Client (NaCI) to the enclave environment, thereby containing untrusted data-processing modules to prevent any leakage of user input data. The solution comes with a verifier and a service runtime. The verifier disassembles the binary and validates that the disassembled instructions are safe to execute, which guarantees that the untrusted module cannot break out of NaCI's SFI sandbox.

Deflection [279] is a model using out-of-enclave instrumentation for in-enclave information-flow control. Deflection uses proof-carrying code (PCC), a technique that enables a verification condition generator to analyse a program and create a proof that attests to the program's adherence to policies, and a proof checker to verify the proof and the code. Deflection builds the binary code for a data-processing program and enhances it with security annotations for real-time policy enforcement. Besides, a trusted code consumer operates within the bootstrap enclave to verify whether the target code genuinely includes the required security annotations.

The main drawbacks of these works are performance and usability. Ryoan is particularly intrusive on the applications' binary due to its security annotations. This entails a substantial overhead, up to 100% and 32% for Royan and Deflection, respectively. Besides, these two tools cannot run legacy code in TEEs as they require the manual partitioning typical of the SGX SDK.

6.3 Twine design

We aim to propose a runtime for running legacy Wasm-compiled applications in secure enclaves, establishing a two-way sandbox that ensures standard TEE security features while mitigating data leaks, even from malicious TEE service providers. The runtime must provide dedicated support to Wasm modules for secure execution of essential OS services within the enclave.

6.3.1 Requirements elicitation

We identify the five requirements that guide the design and implementation of TWINE. The upcoming sections further explain how the architecture of TWINE complies with them.

- $\mathbf{R}_1$. **Must support execution of legacy software services in the two-way sandbox**: TWINE ensures transparent execution of services whose source code can be compiled to Wasm.
- $\mathbf{R}_2$. **Must support communication in the two-way sandbox**: Wasm modules in secure enclaves communicate via selected application-level protocols or secured socket API.
- $\mathbf{R}_3$. **Must support file system in the two-way sandbox**: Wasm modules in secure enclaves perform file system operations based on runtime-defined permissions or access paths.
- $\mathbf{R}_4$. **Must prevent covert channels via OS services**: TWINE protects against sensitive data exfiltration from the enclave, which may occur through OS service interfaces.
- $\mathbf{R}_5$. **Must be verifiable**: TWINE provides support for verifying the integrity of the runtime, the SGX's TCB and validating its related security policies for remote actors.

6.3.2 Threat model

TWINE leverages the protection of TEEs to offer a trusted environment for running Wasm-compiled applications. Many guarantees offered by TWINE are inherited from the underlying TEE implementation, specifically Intel SGX in our proposal. We note that a different TEE implementation may withstand a different level of threats.

Assumptions Adversaries may have physical access to the computer hardware. However, due to the Intel SGX's hardware-intrusion defences, hardware attacks are deemed impractical. The TEE delivers a protection that aligns with Intel's claims, and standard cryptographic techniques cannot be subverted. Enclave codes present no vulnerabilities by implementation mistake.

SGX enclaves Code and data inside enclaves as well as Intel SGX's TCB are trusted, while outside components are considered untrusted. The non-enclaved part of a process, the OS and any hypervisor are thus potentially hostile. The processor ensures that only the enclave can access its memory, which reads encrypted from the outside otherwise. Writing the memory enclave from the outside causes the enclave to be terminated. Side-channel, denial-of-service, fork, or reboot attacks may exist, and applications running inside enclaves must be written to be resistant to them. While we consider these attacks out of scope, mitigations exist [290, 291].

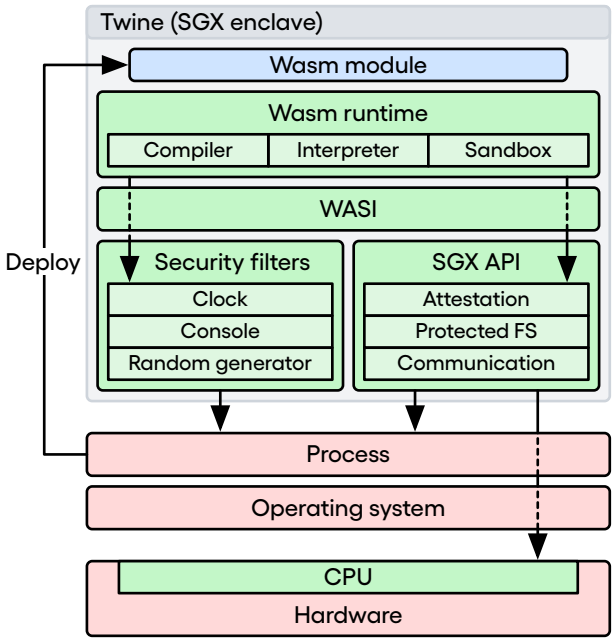


Figure 6.1: The TWINE architecture. The green boxes (▢) are the TCB of Intel SGX. The red boxes (▢) are untrusted. The blue box (▢) is the Wasm module that is attested.

Operating system The OS follows an honest-but-curious threat model, and it conforms to its specification, posing no threat to user and kernel processes. A compromised OS may arbitrarily respond to enclave calls, causing its malfunction. Enclaves must be carefully crafted to ignore abnormal responses or even abandon execution in such cases.

Wasm The Wasm specifications and implementation (*e.g.*, sandbox, virtual machine) are sane and do not present vulnerabilities. Side-channel attacks are beyond the scope of this work [292].

Two-way sandbox In the TWINE runtime, the threat model for hosted applications is based on an honest-but-curious adversary. The first layer of defence relies on Wasm and WASI sandboxing to mitigate unauthorised code execution and data modification, shielding the infrastructure owner. In contrast, the second layer leverages Intel SGX to counter unauthorised data access and code tampering, aimed at securing the application and data owner. We further analyse the security implications of AOT-compiled Wasm code in Section 6.7.

6.3.3 Architectural overview

TWINE comprises two main building blocks: the Wasm runtime and the WASI interface, as detailed in Figure 6.1. The Wasm runtime runs entirely inside the TEE, and WASI acts as a bridge between trusted and untrusted environments, abstracting the machinery dedicated to communicating with the TEE facilities and the underlying OS. The TEE-hosted Wasm runtime uses the WASI interface, which is always involved whenever OS services are accessed, allowing it to filter security-sensitive OCALLs. Doing so serves as an intermediate control layer preceding

the actual interaction with the OS, adhering to a capability-based security model. The runtime can limit what Wasm can do on a program-by-program basis, hence enforcing the sandbox, which prevents Wasm code from interacting with the underlying OS. For instance, WASI may restrict an application to open a file system subtree only, similar to the capabilities of *chroot*. The combination of the sandbox of SGX and WASI ends up in a two-way sandboxing system, partially inspired by MiniBox [293], and somehow follows the same perspective as Ryoan. The system, considered untrusted in SGX's threat model, cannot compromise the integrity of the enclave code or the confidentiality of the data stored in memory. Likewise, Wasm-compiled applications, considered untrusted from the data owner's standpoint, cannot interact directly with the OS unless WASI explicitly grants permission in the Wasm runtime.

Compliance with R_1 TWINE facilitates the porting of legacy applications to be executed within TEEs for three primary reasons. Firstly, the programming language can be freely chosen, provided it can be compiled with LLVM or another compiler that supports Wasm and WASI as a compilation target. This lifts the limitations related to the language support of the official SDK of SGX, which is only provided for applications written in C/C++. Secondly, it is cross-platform hardware-compatible. Applications can be safely executed as long as the TEE is able to execute Wasm (supported by WASI), opening the door to other TEE implementations. Finally, it is system-agnostic as long as the OS can provide an equivalent of the API required by WASI. Since WASI mimics the system calls of POSIX systems, many Unix-like variants can implement it.

Compliance with R_2, R_4 We have appropriately instrumented the WASI and the runtime to control security-sensitive functionalities, which could be exploited to exfiltrate data. Depending on the runtime configuration, low-level communication functionalities (*e.g.*, socket-related OS calls) may not be provided directly to the Wasm binary. Instead, the runtime may provide network communication through application-level protocols (*e.g.*, HTTP) to prevent covert channels via network system call interfaces. To this end, we embedded an HTTP library inside as a Wasm module, which can be configured in terms of whitelisted targeted endpoints.

Compliance with R_3, R_4 File management support is provided and shielded against data leaks from code running in the Wasm module. WASI has been extended so that every file operation leverages the Intel Protected File System (IPFS), which automatically encrypts data coming out of the two-way sandbox (*e.g.*, via `fwrite`) and decrypts data flowing in the other direction (*e.g.*, via `fread`). Furthermore, the WASI sandbox enforces limitations on the Wasm applications by restricting them to predefined file system operations and access paths.

Compliance with R_5 TWINE incorporates an attestation service that enables data owners to verify the authenticity and integrity of the SGX's TCB, the Wasm-compiled application, and the trusted runtime, including WASI, through which their data flows. Wasm applications can interact with an embedded attestation library to issue evidence that are notably bound to the hashes of the Wasm-compiled application, the runtime, and the security policies controlling how the configuration of WASI. Data owners rely on such attestation mechanisms to confidently trust that remote applications are secure, verifiable, and that the runtime operates according to the specified security policies, preventing confidential data to leave the enclave.

Confidentiality of enclave code Using the conventional approach for programming and deploying Intel SGX enclaves, the TEE guarantees the confidentiality and the integrity of the data handled at runtime. However, Intel SGX does not feature the confidentiality of the enclave's code, but only its integrity. While integrity is verified through a signature of the code, the code itself must remain in clear text to be loaded into the enclave memory. Extensions, such as Intel SGX protected code loader (PCL), extend confidentiality for enclave binaries, albeit with some security considerations, including writable code sections. These considerations may result in editable enclave code and read-only data at enclave runtime [294]. Conversely, TWINE can also offer code confidentiality for enclave binaries without the limitations associated with PCL. Wasm modules can be downloaded into the enclave following attestation or retrieved from a sealed blob and subsequently loaded by TWINE. Upon decryption of the Wasm module, it is mapped into a secure memory area of SGX known as the *reserved memory*. The runtime can load arbitrary executable code into this memory region, and due to the inherent robustness of Wasm, such code remains immutable from the perspective of the Wasm application.

6.4 Implementation details

As presented in Section 4.4, we considered many Wasm runtimes as candidates for implementing TWINE. We have chosen WAMR for its small size, few dependencies, and its ability to be linked to binary code (albeit generated ahead of time, that is, no JIT compilation). A small TCB reduces the attack surface of the runtime. Furthermore, AOT-compiled Wasm applications achieve near-native execution speed, as shown later, in the evaluation of TWINE. As such, we forked WAMR and extended its WASI implementation, as explained below, in such a way that we can abstract the enclave constraints while providing systems calls.

WASI is the interface through which Wasm applications communicate with the outside world, similar to POSIX's capabilities for regular native programs. Developing TEE-enabled applications requires crossing the boundary between trusted and untrusted environments, materialised with ECALLs and OCALLs in the case of Intel SGX. Leveraging WASI as the communication layer meets the purpose of Wasm, where the implementation is abstracted away for the application itself. As a result, the applications compiled in Wasm with WASI do not require any change to be executed inside Intel SGX or other TEE implementations. For example, WATZ, introduced in Chapter 7, showcases how the same Wasm applications can be hosted in Arm TrustZone.

By the time TWINE was developed, WAMR already included a WASI implementation that relies heavily on POSIX calls. POSIX is not available in SGX enclaves, so the WASI layer written by the authors of WAMR needs to cross the trusted boundary of the enclave frequently and routes most of the WASI functions to their POSIX equivalent using OCALLs. While this approach enables running any Wasm applications that comply with WASI in an enclave, it does not bring additional security benefits regarding the data that transits through POSIX, as there is no encryption.

We designed TWINE with a more optimised WASI implementation for WAMR, better tailored to SGX enclaves, which adopts a different approach than plain forwarding WASI calls outside the enclave. The rationale for this choice is as follows. First, performance: most WASI calls

would simply be translated to (costly) OCALLs. Second, we wanted to leverage trusted implementations when available. Therefore, we refactored WAMR's WASI implementation to keep its sandboxing enforcement, splitting the remaining into two distinct layers: (i) one for specific implementations when available and (ii) another for generic calls. Generic calls are handled by calling the POSIX library outside the enclave while providing additional security measures and validity checks. Such calls are only wired when no trusted compatible implementation exists. For instance, Intel SGX does not support time retrieval. Hence, TWINE's WASI layer fetches monotonic time while ensuring that the returned values are always greater than the previous ones. If, for a given function, a corresponding trusted implementation exists (as it is the case for the ones in the official Intel SGX SDK), we use it to handle its related WASI call. Often, a trusted implementation calls outside the enclave while at the same time providing additional security guarantees, such as with IPFS which leverages encryption (detailed in Section 6.4.3). Finally, TWINE can disable untrusted POSIX implementations inside the enclave (via a compilation flag). This is useful when one requires a strict and restricted environment or assesses how the applications rely on external resources. In particular, WASI may expose states from the TEE to the outside by leaking sensitive metadata in host calls, *e.g.*, usage patterns and arguments, despite the returned values being checked once retrieved in the enclave.

In its current implementation, TWINE requires exposing a single ECALL to supply the Wasm application as an argument. This function starts the Wasm runtime and executes the start routine of the Wasm application, as defined by the WASI ABI specifications [295]. TWINE is versatile and can be adapted to only receive the Wasm applications from trusted endpoints supplied by the applications providers. The endpoint may either be hardcoded into the enclave code and, therefore, part of the SGX measurement mechanism that prevents binary tampering, or provided in a manifest file with the enclave. The endpoint can verify that the code running in the enclave is trusted using remote attestation. We propose a remote attestation API for Wasm applications in Section 6.4.4. As a result, TWINE is a secure deployment and execution framework for running applications on untrusted devices and environments. Despite OS dependency for network communication, TWINE provides cryptographic techniques to create TLS and HTTPS channels that cannot be eavesdropped on. For that purpose, we compiled and integrated a Wasm cryptographic library in the runtime (detailed in Section 6.4.2).

6.4.1 Memory allocation

Memory management greatly impacts the performance of the code executed in enclaves (see Section 6.6.2 and 6.6.4.1). WAMR supports three modes to manage the memory for Wasm applications: (i) the default memory allocator of the environment (*e.g.*, `malloc` and `free` for Linux), (ii) a user-defined memory allocator, and (iii) a preallocated buffer of memory. We found that using the SGX memory allocator to enlarge the linear memory of the Wasm application performed poorly, leading to a time complexity above linear. Consequently, TWINE preallocates a buffer of a fixed size to operate. This approach is generally not problematic, as it requires specifying a fixed heap size at compile-time for SGX enclaves.

We note that Intel SGXv2 include a memory allocation scheme called Enclave Dynamic Memory Management (EDMM) [157], enabling more memory allocation than the size specified at build

time. In such cases, TWINE can leverage this new feature to extend the preallocated buffer dynamically, hiding such implementation details to the Wasm application.

6.4.2 Communication support

Bringing network functionalities inside enclaves is essential to support communication with external endpoints, such as trusted peers or other enclaves. Therefore, we implemented the required calls to deal with network sockets in our WASI layer, relying on the network stack of the untrusted OS. We perform our cryptography inside the enclave to secure the communication channels and prevent attackers from eavesdropping. For that purpose, we use WolfSSL [296], an open-source popular cryptographic library. WolfSSL supports mainstream ciphers and the TLS protocol, which can be used to set up trusted communication channels. Using a renowned cryptographic library compiled in Wasm has many advantages: (i) the library is platform-independent and reusable in other TEEs (*e.g.*, TrustZone with WATZ in Chapter 7), (ii) the library can be statically linked to any application when compiled into the Wasm format, and (iii) the library can also leverage the multi-module feature of WAMR, the runtime which TWINE is based on, which enables Wasm applications to load dependencies at runtime, which eliminates the burden of static linking, and abstracts a specific implementation of the library. A WASI proposal already exists to bring cryptography to Wasm applications by the runtime [270]. However, we considered its status too preliminary to be considered as a building block.

Furthermore, we adapted Mongoose [297], a lightweight and embeddable web server library, to enable compilation into Wasm and facilitate the hosting of web applications within the TEE. In conjunction with WolfSSL, our adaptation of Mongoose enables clients to establish secure communication channels featuring HTTPS termination secured by Intel SGX. Consequently, enclaved applications can expose a high-level API, such as REST, while ensuring the confidentiality of data and the integrity of executing code. The literature has examined various approaches to providing attestation in conjunction with high-level API supported by TLS. Such mechanisms are covered in Chapter 8. TWINE can reuse similar solutions for TEE attestation.

The WASI calls related to the sockets are implemented using OCALLs, forwarding them to the untrusted OS. Analogous to the WASI implementation for Linux, our approach supports the sandboxing of networking, allowing the runtime to supply IP ranges to which Wasm applications can connect. We minimised the number of OCALLs by embedding computations that do not require untrusted OS system calls, *e.g.*, text to binary IP address conversions. WolfSSL has been slightly adapted to be compiled in Wasm. Our work builds upon the compilation target of WolfSSL for Intel SGX, with missing dependencies addressed using WASI calls and header files of the WASI-SDK supporting C/C++. Due to the constraints of the Wasm virtual machine, which prohibits embedding assembly instructions in the bytecode, we could not use the hardware acceleration offered by modern CPUs for specific ciphers (*e.g.*, AES). Limitations can be mitigated by offloading certain cryptographic operations to the runtime. Mongoose exclusively supports OpenSSL and Mbed TLS libraries to provide cryptographic primitives. As such, we integrated WolfSSL as a TLS provider within Mongoose for hosting or querying HTTPS websites.

6.4.3 File management support

As a showcase of the abstraction offered by WASI, we implemented the subset of the WASI calls related to file system operations by using the IPFS [280]. Being shipped with the Intel SGX SDK, it mimics the POSIX functions for file operations. The architecture of IPFS is split in two: (i) the trusted library, running in the enclave that offers a POSIX-like API for file management, and (ii) the untrusted library, an adapter layer to interact with the POSIX functions outside the enclave, that read and write on the file system. Upon calls to `write`, content is encrypted transparently by the trusted library before being written on the media storage from the untrusted library. Conversely, content is decrypted by the enclave during calls to `read`.

IPFS uses AES-GCM for authenticated encryption, leveraging the CPU's hardware acceleration. An encrypted file is structured as a Merkle tree with nodes of a fixed size of 4 KiB. Each node contains the encryption key and tag for its children nodes. Thus, IPFS iteratively decrypts parts of the tree as the program in the enclave requests data [298]. This mechanism ensures the confidentiality and integrity of the data stored on the untrusted file system. While the enclave is running, SGX's memory shielding guarantees the confidentiality and integrity of the data.

IPFS has several limitations, which are deemed beyond the scope of Intel's threat model. Since the files are saved in the regular file system, there is no protection against malicious file deletion and swapping. Consequently, IPFS lacks protection against: (i) rollback attacks: IPFS cannot detect whether the latest version of the file is opened or has been swapped by an older version, and (ii) side-channel attacks: IPFS leaks file usage patterns and various metadata such as the file size (up to 4 KiB granularity), access time and file name. We note how Obliviate [69], a file system for SGX, partially mitigates such attacks. Although Obliviate can be adapted for use with TWINE, addressing side-channel attacks falls beyond our threat model.

WASI includes several calls that do not have direct counterparts in the IPFS, due to slight variations from the ISO C standard. For instance, the function `fseek` allows the cursor to move past the end of a file, which is not permitted in IPFS. To address this discrepancy, our WASI implementation extends the file with `null` bytes, requiring extra IPFS calls. Also, IPFS lacks support for vectored read and write operations. Since WASI exclusively handles file I/O with vectored operations, we resolved to implement those with an iteration.

IPFS provides convenient support to automatically create keys for encrypting files, derived from the enclave signature and the processor's (secret) keys. While automatic key generation seems straightforward, a key generated by a specific enclave in a given processor cannot be regenerated elsewhere. IPFS circumvents this limitation with a non-standard file open function, where the caller passes the key as a parameter. Our prototype relies on automatic generation, and we leave it as future work to extend our WASI layer to support custom encryption keys.

In conclusion, files persisted by TWINE cannot be read outside the enclaves and are transparently decrypted and integrity checked while handled by Wasm applications.

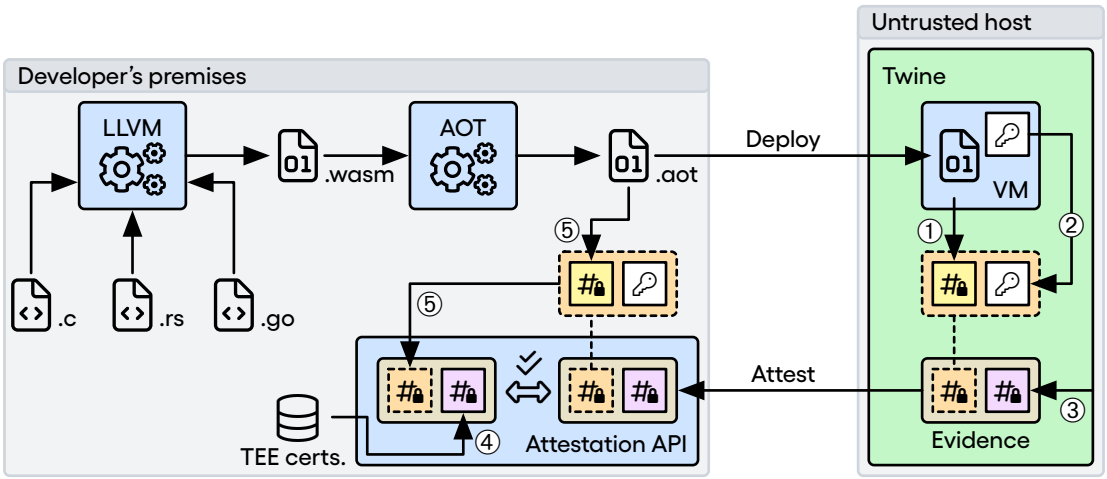


Figure 6.2: The deployment and attestation workflow of Twine. The blue boxes (▢) are software components, the green box (▢) is the TCB of Intel SGX, the white boxes (▢) are the key for a secure communication channel, the yellow boxes (▢) are the measurement of the enclave, the brown boxes (▢) are the evidence and the pink boxes (▢) are the signature of Intel SGX.

6.4.4 Attestation

Remote attestation, previously covered in Chapter 3, is a cornerstone feature of TEEs, as it ensures the authenticity of the executing code, including Wasm applications in the context of Twine. We worked with the open-source community of WAMR [299, 300] to define additional functions in the runtime to interact with the attestation features of Intel SGX. As a result, Wasm applications within our system can interface with Intel SGX to generate evidence during attestation. The integration of attestation within the runtime provides robust security guarantees for remote peers, typically facilitating the establishment of secure communication channels to transfer confidential data in remotely executed Wasm applications.

Figure 6.2 depicts the workflow of Twine, including the deployment and attestation of Wasm applications. The source code from potentially multiple programming languages is compiled to Wasm bytecode, followed by AOT compilation to native code for enhanced performance. The resulting binary is then deployed into Twine's enclave. Remote attestation is implemented using `librats`, a low-level library facilitating attestation for multiple TEEs [137]. Although the current implementation supports Intel SGX Data Center Attestation Primitives (DCAP), the runtime can be extended to support additional TEEs. A hash is computed upon loading the Wasm application bytecode (or the AOT-compiled assembly code) within the enclave (①). When collecting a quote, the runtime retrieves this precomputed hash and derives a secondary hash, comprising optional data provided by the Wasm application, such as a public key to establish a trusted communication channel (②). The runtime then forwards the final value to the Intel SGX RA mechanism, which issues evidence (③). Relying parties may use evidence to confirm the enclave's genuineness (④) and the trustworthiness of the executing Wasm application (⑤).

6.5 Demonstrating Twine with a distributed financial technology

The company *Credora* provides a privacy-preserving scoring solution for credit in cryptocurrency finance. *Credora*'s requirements is that users' confidential data remains private and is computed correctly, *i.e.*, metrics must reflect the actual status of the credit. To this end, *Credora* uses TEE and cryptographic proofs to ensure users' sensitive data privacy and guarantee risk analysis. Intel SGX is the TEE implementation adopted to protect and attest sensitive processing. Within the threat model supported by Intel SGX, only computations authorised by the user are allowed in the attested enclave, and no party can see granular private data or perform any knowledge extraction. The Twine runtime has been used in this data-intensive, large-scale commercial application to enhance the trustworthiness of a distributed credit scoring oracle. *Credora* uses Twine for easing the development, deployment and attestation of Intel SGX applications.

Credit scoring has been historically used by financial institutions to estimate the risk of lending money to an individual. It determines the ability of an entity to repay debts based on a number of quantitative and qualitative metrics. High credit scores lead to higher chances of obtaining a loan with low interest rates, while lower score result in higher interest rates. *Credora* provides a privacy-preserving scoring solution for credit in cryptocurrency finance. With over $100 B of crypto-collateral being used to generate over $1.25 B of interest on a quarterly basis, credit is one of the most rapidly growing sectors of the emerging cryptocurrency finance ecosystem. *Credora* allows borrowers to supply lenders with real-time portfolio risk metrics, while preserving the privacy of trades, positions, and other sensitive information. Borrowers benefit from improved lending terms, as they can display their risk in real-time and assure lenders they are trading responsibly. Lenders benefit from increased visibility and real-time information. *Credora* calculates various dynamic risk metrics and aggregates across client portfolios, including total assets, liabilities and maximum loss simulations. The latter is based on the standard portfolio analysis of risk (SPAN) system, wherein the worst possible loss of a client's portfolio is estimated from a simulation over price and volatility shock scenarios.

Figure 6.3 shows the architecture of the credit scoring system of *Credora*. It is composed of a set of distributed and loosely-coupled microservices communicating via a distributed in-memory data store. The typical execution flow is as follows. Initially, before interacting with the backend, the client challenges the key management system (KMS) for its attestation (①) to set up a secure channel. Once completed, it can provide secrets (*i.e.*, *exchange keys*) to the *Credora* TEE-secured KMS (②). The *dispatcher* defines pulling jobs, *i.e.*, a request to be executed for a particular exchange using its API endpoints that returns information on clients' portfolios. These jobs are transferred through a shared cache to the pullers (③). Based on those, the *private puller* uses the *exchange keys* distributed by the KMS (④) to get clients' data (*e.g.*, information on open trading positions) from cryptocurrency exchanges venues (⑤) such as *Binance*, *Deribit*, *Coinbase* and *Kraken*. A trading position denotes an individual's or entity's ownership stake in a particular financial asset, which represents their investment and potential for profit or loss. The obtained data is encrypted and pushed into the shared cache (⑥). Similarly, the *public puller* gathers public market data (⑤), which is also stored in the shared cache (⑥). The *aggregator* reads clients' private data from the shared cache, markets public data obtained by the *public puller* (⑦), and computes the risk metrics and credit scores (⑧).

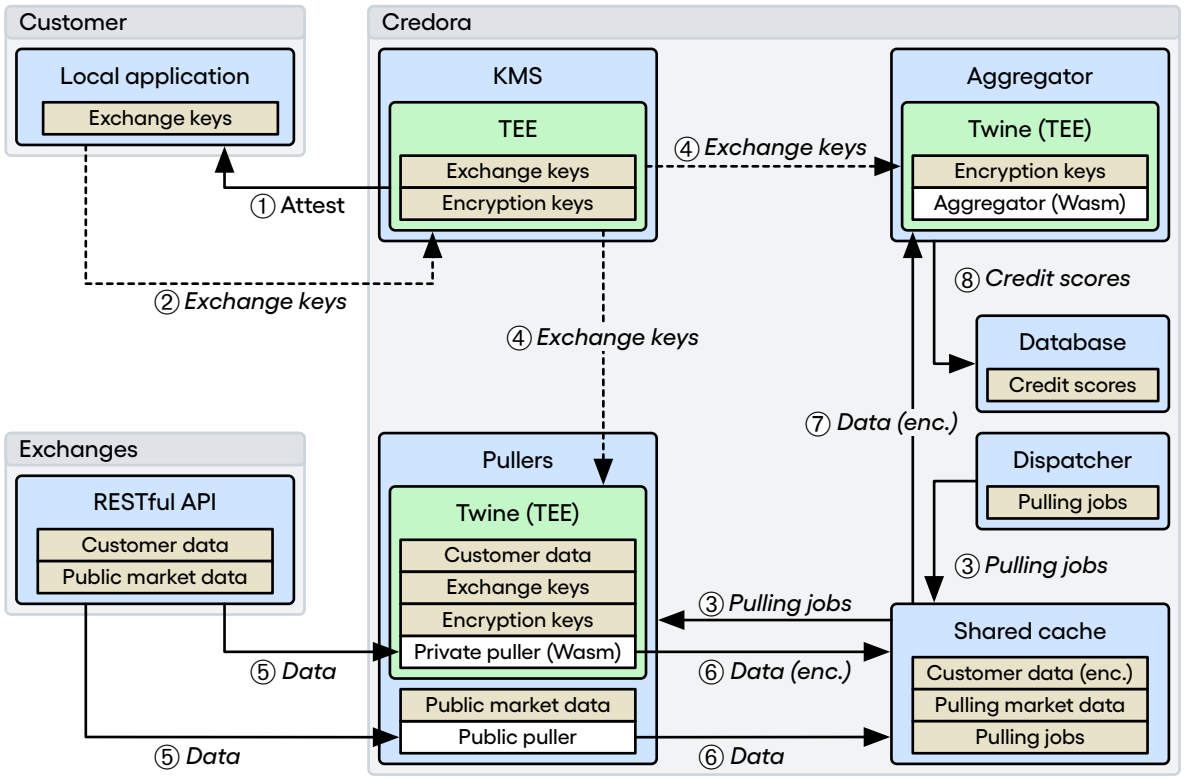


Figure 6.3: The architectural overview and workflow of *Credora*. The solid arrows describe trusted channels, whereas dashed arrows are attested channels. The blue boxes (▢) are organisational units, the green boxes (▢) are TEEs (some with Twine), the brown boxes (▢) are data, and the white boxes (▢) are software applications.

In this use-case, the confidential data is given by: (i) *exchange keys* used to obtain clients' wallet data from *exchange* venues, and (ii) the client trading positions received from *exchanges*. This architecture guarantees that confidential data never leaves the secure enclave unencrypted. The *exchange keys* are received directly from clients' browsers over an attested TLS TEE-terminated secure channel. This is possible through the *librats* library [137], the same attestation library embedded in Twine for issuing evidence, which can be compiled to Wasm using the compiler Emscripten and executed inside the browser. These *exchange keys* are persistently stored using IPFS that encrypts the secret information in the enclave using the SGX sealing key. The process of signing requests for *exchanges* using the *exchange keys* is executed inside the enclave. RESTful requests for *exchanges* are then made over an HTTPS TEE-terminated connection using WolfSSL and Mongoose compiled in Wasm, which guarantees that the confidential financial data is directly received and aggregated in the enclave.

Under the described scenario, *Credora*'s clients only need to trust Intel (*i.e.*, SGX threat model) and *Credora* itself, which does not disclose the implementation of their software. Hence, the cloud provider hosting the application may be untrusted, thanks to the secure environment offered by Intel SGX. Additionally, we aim to further reduce the scope of the threat model by removing *Credora*, leaving Intel as the only trusted entity. Toward this goal, one must prove to clients that no data has ever leaked from the secure enclaves, nor that the enclaves are

curious and retrieve unwanted information from the cloud provider's system. As such, TWINE's trust model plays an essential role: its two-way sandbox (as explained in Section 6.4) offers two strong guarantees for the cloud provider and the clients. First, the *Credora*'s enclaves, which are based on TWINE, are proven authentic using remote attestation, which guarantees to *Credora* that the enclaves have not been tampered with and can supply Wasm applications and confidential information for further computations, preventing one from eavesdropping. Second, *Credora*'s customers can review the open-source implementation of *Credora*'s enclaves (*i.e.*, based on TWINE), ensuring that the runtime properly sandboxes the Wasm applications (whose source code is proprietary) deployed by *Credora* later on, which prevents the *Credora*'s Wasm applications from accessing the host system resources.

6.6 Evaluation

We present here our extensive evaluation of the runtime TWINE. We intend to answer the following questions: (i) What are the performance overheads of using TWINE on Client and Scalable SGX, compared to native applications and AccTEE, a state-of-the-art solution? (ii) What are the performance overheads for using cryptographic operations and setting up TLS-terminated connections within the enclaves? (iii) Can a database engine be compiled into Wasm and executed in a TEE while preserving acceptable performance? (iv) How do the database input and output operations behave when the EPC size limit is reached? (v) What primitives generate most of the performance overheads while executing database queries? Can we improve them? (vi) How does TWINE perform when used in a data-intensive and real-world solution?

We answer these questions by using a general-purpose compute-bound evaluation with PolyBench/C (§6.6.2), encrypting, hashing and securing communications using cryptographic primitives and TLS with WolfSSL (§6.6.3), evaluating a general-purpose embeddable database using SQLite (§6.6.4), stressing the database engine using custom micro-benchmarks that perform read and write operations (§6.6.4.1), analysing various cost factors bound to Wasm and SGX (§6.6.4.2), profiling the time breakdown of the database components, the Wasm runtime and the SDK of SGX (§6.6.4.3), and finally assessing the end-to-end performance of a Wasm application in the fintech company *Credora* (§6.6.5).

6.6.1 Experimental setup

We use a Supermicro 5019S-M2 with Intel Xeon E3-1275 v6 (3.8 GHz, 128 MiB of Enclave Page Cache (EPC), usable 93 MiB) for Client SGX tests, and a Supermicro SYS-520P-WTR with an Intel Xeon Gold 6326 (2.9 GHz, 8 GiB of EPC) for Scalable SGX. The benchmarks are executed using Client SGX, except where noted. Systems run Ubuntu 18.04.6 with kernel 4.15.0, SGX driver v2.11, and SGX SDK v2.17.100.3. TWINE is fully merged into the WAMR's official repository. As such, we use the WAMR build for running the benchmarks unless stated otherwise.

Time is measured using the monotonic clock of POSIX in all the benchmarks and averaged using the median. If measured from within the enclave, the time to leave and reenter the enclave is included. In our setup, the enclave round trip accounts for approximately 4 ms. We

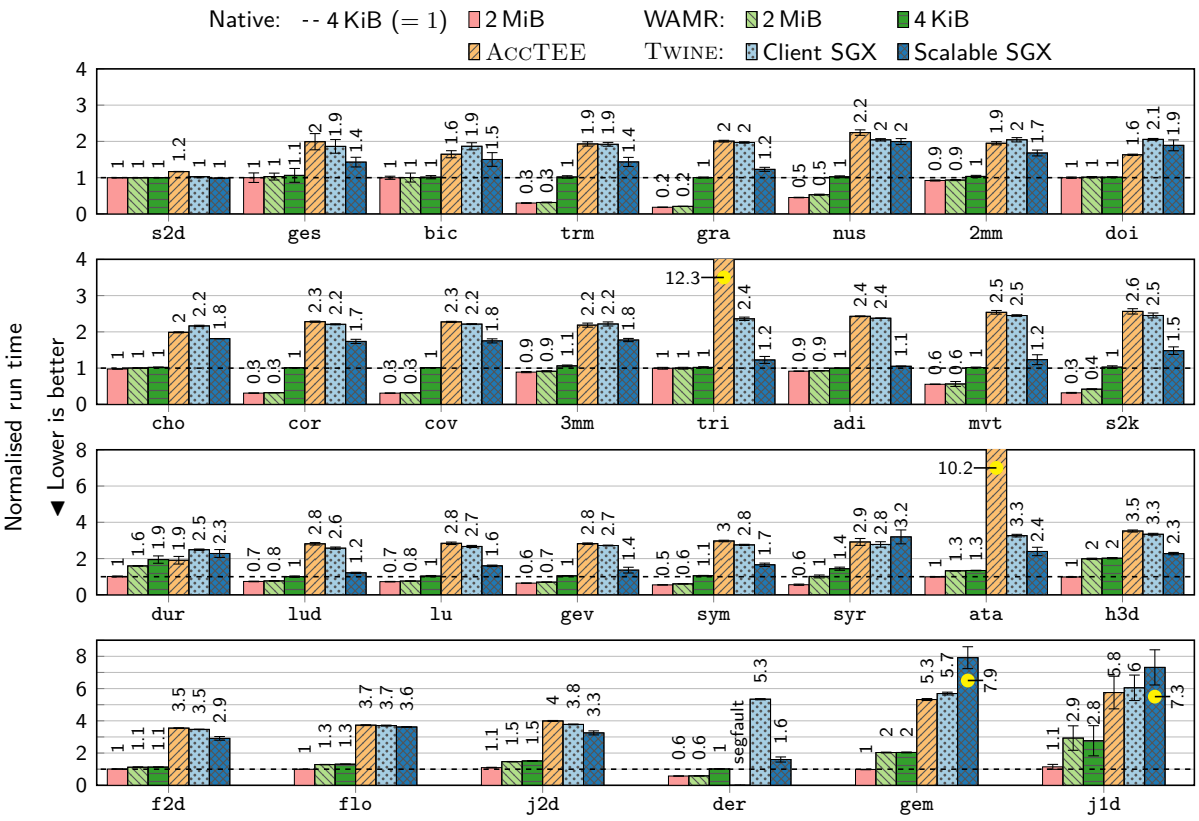


Figure 6.4: The performance of PolyBench/C benchmarks, normalised to native speed, with memory page sizes of 4 KiB and 2 MiB.

used Docker to build the benchmarks while their execution is on bare metal to avoid potential isolation overheads. The native benchmarks are compiled using Clang 10 with optimisation set to `--O3`. The Wasm benchmarks are compiled using Clang into Wasm format, then AOT-compiled into native format using the compiler provided by WAMR (*i.e.*, `wamrc`) using `--O3` and size level 1 to run into SGX enclaves (`--sgx`). We used GCC v7.5 for two tasks: (i) compile the applications executing the benchmarks, *i.e.*, the WAMR runtime and the SGX enclaves, also with `--O3`, and (ii) compile IPFS with `--O2`, as in the SGX SDK. SGX-LKL (v0.2.0), LKL (v5.4.62) and AccTEE have been used as an empirical baseline for running the experiments natively in SGX enclaves. They have been downloaded from the official Debian repository and GitHub. Finally, our implementation and instructions to reproduce our experiments are open-source [15].

6.6.2 Micro-benchmarks: PolyBench/C

PolyBench/C [301] is a CPU-bound benchmark suite commonly used to validate compiler optimisations and compare the performance of Wasm execution environments [116, 120]. We leveraged PolyBench/C due to the practicality of deploying it in SGX enclaves. We present results for 30 PolyBench/C (v4.2.1) tests: native (x86-64 binaries), AOT-compiled Wasm using WASI-SDK for WAMR and TWINE, and JIT-compiled using V8 for AccTEE. We also benchmark

TWINE with Client and Scalable SGX, highlighting performance overheads due to the EPC limit. For accurate precision when compared to AccTEE, a state-of-the-art runtime with resource accounting, we disabled the instrumentation of the Wasm bytecode. Furthermore, we benchmarked native and WAMR with two sizes of memory pages: 4 KiB and 2 MiB. Intel SGX-enabled solutions omit large memory pages due to the absence of support in SGX. Figure 6.4 shows the results normalised against the native run time using memory pages of 4 KiB.

We can split the PolyBench/C test results into four groups based on the performance ratio of the execution modes (native, WAMR and TWINE) and the memory page sizes (4 KiB and 2 MiB): (i) similar execution time (*e.g.*, `s2d`); (ii) WAMR, AccTEE and TWINE (*i.e.*, the Wasm runtimes) are slower than native (*e.g.*, `h3d`, `gem` and `j1d`); (iii) AccTEE and TWINE (*i.e.*, the Wasm runtimes which leverage Intel SGX) are slower than native and WAMR (*e.g.*, `ges`, `bic` and `doi`); and (iv) native and Wasm using large memory pages are faster than native and Wasm using regular memory pages (*e.g.*, `trm`, `gra` and `cor`).

While Wasm may be up to speed with native performance, Wasm is usually slower than native for several reasons, such as increased register pressure, more branch statements, increased code size, and high reliance on the stack. We also looked at the impact of memory on performance, given the additional cost for SGX enclaves [62]. Beginning with 160 MiB, the minimum for all PolyBench/C tests, we incrementally reduced the Wasm runtime memory allocation until allocation failure. The slowdown in the `der` test is due to reaching the EPC size limit, while AccTEE throws a segmentation fault unless the array dimensions are reduced (as in the original paper). Similarly, `lu` and `lud` require at least 80 MiB of memory. In our settings, AccTEE showed spikes in performance for the `tri` and `ata` tests. Despite higher normalised run times compared to TWINE, the absolute run times for these tests stay under one second.

We further examine the overheads caused by the page evictions from the EPC. For that purpose, we run PolyBench/C on TWINE with Scalable SGX, benefiting from a larger EPC of 8 GiB. Memory-intensive tests, such as `der`, `lu`, and `lud`, reveal fewer overheads compared to Client SGX. Besides, the majority of tests show reduced overheads, even if they did not use the total EPC capacity. A possible reason for this performance difference is Scalable SGX's transition to Intel's Total Memory Encryption (TME), which uses AES-XTS, over the traditional Merkle tree that maintained the integrity of enclaves (detailed in Section 2.2.1) [164]. While this choice downgrades SGX's threat model by relaxing defences against hardware integrity and anti-replay attacks, this trade-off offers larger EPCs and generally more efficient enclave execution [160].

We also analysed the usage of large memory pages for two reasons. First, enabling large pages for a particular process that has yet to be tailored for using them usually requires turning on the OS-wide feature *Transparent Huge Pages*, impacting all the other processes running on the same system. On the other hand, TWINE abstracts the memory allocator as a process supporting this feature, which transparently enables any given Wasm program to allocate large pages. Second, PolyBench/C highlights that Wasm memory-intensive workloads with large pages enabled may be faster than their native counterparts when not configured for using this feature, offering an effortless performance boost. This speedup is due to the reduced pressure on the processor translation lookaside buffer (TLB) cache, which translates virtual to physical memory addresses.

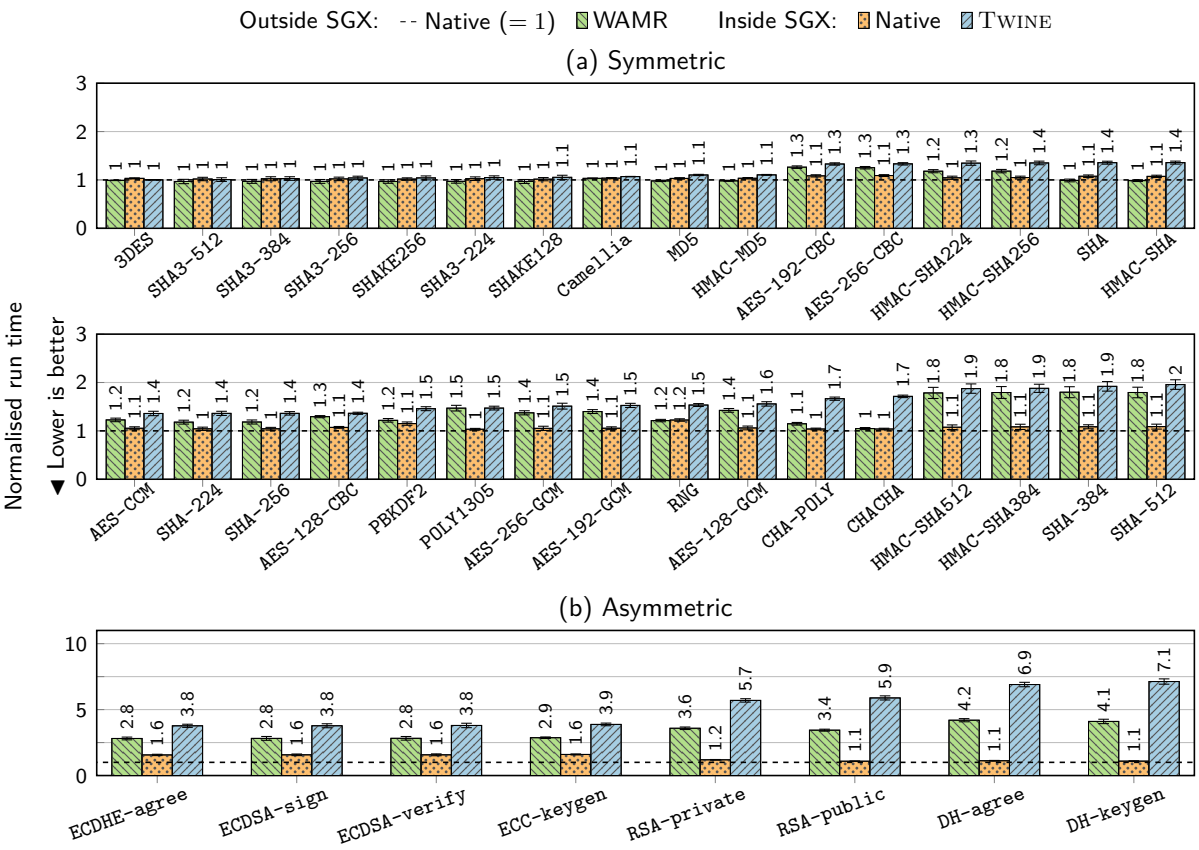


Figure 6.5: The performance of WolfSSL benchmarks targeting cryptographic algorithms, normalised to the native speed.

6.6.3 Micro-benchmarks: network stack

We assess the network stack of TWINE using two micro-benchmarks bundled with WolfSSL: (i) a performance comparison of cryptographic primitives using many ciphers and hashing algorithms, and (ii) a performance evaluation of TLS sessions, which have one side of the connection terminated within the enclave. In both cases, we compare the relative execution speed of native and Wasm (inside SGX) and Wasm (outside SGX) against native (outside SGX). For a fair comparison between native and Wasm, we have chosen to disable the hardware acceleration support of WolfSSL, *i.e.*, the offloading operations to the CPU's instruction set architecture (ISA).

Figure 6.5a depicts the execution speed of symmetric algorithms and hashing functions. In contrast, Figure 6.5b illustrates the speed for asymmetric ciphers and key generation operations. Among these, symmetric operations are the most efficient, with an average slowdown for Wasm of 1.2× outside SGX, and 1.3× inside the TEE, compared to native speed execution. Hashing functions follow, with slowdowns of 1.3× and 1.4× for Wasm outside and inside SGX, respectively. Finally, the asymmetric ciphers have the highest slowdown, with 3.3× and 5.1× for the same settings. Despite the slower speed of asymmetric ciphers for session establishment, au-

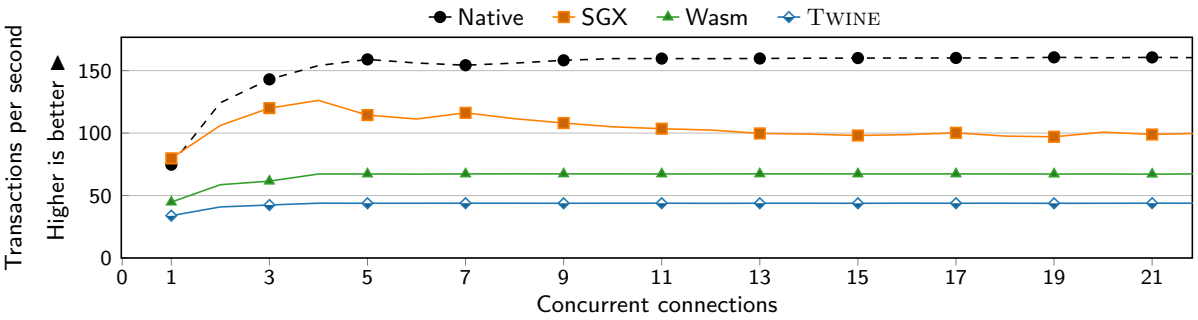


Figure 6.6: The performance of WolfSSL benchmarks for TLS sessions.

thentication, and key exchange, the TLS protocol mitigates their performance impact by relying on more efficient symmetric and hashing algorithms for the remainder of the session.

In Figure 6.6, we stressed an application using WolfSSL's TLS 1.3 protocol by evaluating the number of TLS transactions per second over a range of concurrent connections. The setup involves a client and a server hosted on different machines connected through a switch. The client is a native executable running on Linux, while the server is of four types and evaluated separately: native in Linux and SGX, Wasm in Linux and TWINE in SGX. A TLS transaction is composed of (i) the TLS handshake, (ii) the server reading 16 KiB, (iii) the server sending 16 KiB, and (iv) the closure of the session. We measure the time the server takes to handle 512 connections while varying the number of concurrent connections handled by a single core. Besides, we used the cipher suite `TLS13-AES128-GCM-SHA256` as cryptographic primitives. Finally, we considered `AES128-CCM` and `CHACHA20-POLY1305` as authenticated encryption with associated data (AEAD) ciphers, but we did not notice significantly different results because the performance of these algorithms is similar, as reflected in Figure 6.5.

The number of transactions per second for native execution outside SGX and Wasm demonstrates a pronounced increase until it reaches a maximum value, after which it stabilises. This maximum value means the saturation of the single thread responsible for managing TLS sessions. The native execution within the enclave exhibits a comparable pattern. However, converging after attaining its peak value requires a more extended period. Although we did not conduct a comprehensive investigation on this particular behaviour, we believe the observed phenomenon results from how SGX-LKL (the library OS used to execute native applications in SGX) processes in-enclave packets via its dedicated TCP/IP stack. Native execution outside SGX converges to an average of 157 TPS, which serves as the baseline measurement. In contrast, native execution within the TEE exhibits an average of 97 TPS with a consequent slowdown factor of 1.6×. Furthermore, the average transaction rates per second for Wasm outside and inside the enclave are 67 TPS and 44 TPS, respectively, with corresponding slowdown factors of 2.4× and 3.6×. When comparing the in-enclave solutions, TWINE's slowdown relative to native is 2.2×, but offers all the advantages of Wasm, such as portability and security. Yet, Wasm services hosting TLS connections may leverage multithreading for performance enhancement.

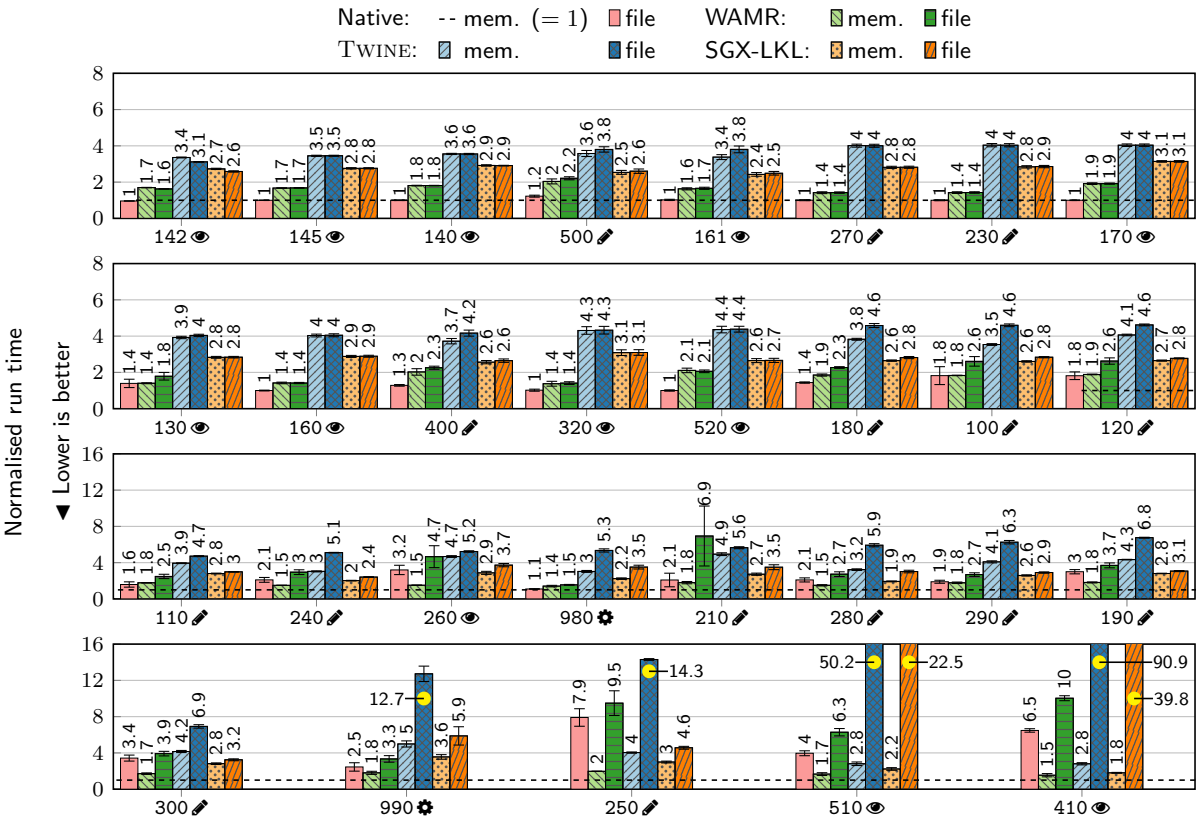


Figure 6.7: The relative run time of SQLite `Speedtest1` benchmarks for reading (👁), updating (✎), and housekeeping (⚙) operations.

We highlight that modifications were made to the official benchmark to use the `poll` system call instead of `epoll`, as the latter is unsupported in WASI. The resulting software was then contributed to the WolfSSL open-source repository [38].

6.6.4 Macro-benchmarks: SQLite

SQLite [302] is a widely-used full-fledged embeddable database. It is ideally suited for SGX, thanks to its portability and compact size. For this reason, we thoroughly evaluated it as a showcase for performance-intensive operations and file system interactions. SQLite requires many specific OS functions missing from the WASI specifications due to standardisation and portability concerns in Wasm. Therefore, we leveraged SQLite's virtual file system (VFS) and implemented a minimal file system interface compatible with WASI to make SQLite process and persist data, reducing the POSIX functions to be supported. We used one of the official templates (`test_demovfs`) to override the OS interface of SQLite since it relies on a few POSIX functions covered by the WASI specifications. SQLite uses a 2048-page cache of 4 KiB each (for a cache size of 8 MiB) with the default (normal) synchronous mode and the default (delete) journal mode. Besides, we use an alternate memory allocator (`SQLITE_ENABLE_MEMSYS3`) to provide a large chunk of preallocated memory for the database instance.

Memory allocation in SGX enclaves can be costly, consuming up to 45% of CPU time in some tests. Preallocating memory can, therefore, significantly improve performance when the database size is predictable. As such, we configured TWINE to allocate a memory buffer of 60 MiB upon runtime instantiation. We compiled SQLite v3.32.3-amalgamation (*i.e.*, a single-file version of the entire SQLite program) into Wasm. First, we used SQLite's own performance test program, `Speedtest1` [303], running 29 out of the available 32 tests, covering a large spectrum of scenarios (we excluded three experiments because of issues with SQLite VFS). Each `Speedtest1` experiment targets a single aspect of the database, such as selection using multiple joints and updating indexed records. Tests are composed of an arbitrary number of SQL queries, potentially executed multiple times depending on the load to generate. Figure 6.7 shows our results, normalised against the native execution. We include results for in-memory configurations as well as for a persisted database, where WASI is used in combination with IPFS.

While we provide additional details below, we observed across all tests that the WAMR's slowdown relative to native on average is 1.7× for in-memory and 1.6× for file-based databases. TWINE's slowdown relative to WAMR is 2.1× for in-memory and 2.0× for file-based databases. Symbols (👁, ✎, ⚙) indicate read queries, data updates (*e.g.*, inserting, updating, and deleting records), and housekeeping tasks (*e.g.*, integrity checks, statistics collection). The upper plot shows a consistent performance penalty trend by variant, with a slowdown of 3.9× and 4.6× for TWINE in-memory and file-based databases, respectively. These experiments primarily represent read queries from the benchmarks. Experiments demonstrating identical performance for in-memory and persistent databases suggest actions on the page cache without file system interaction. Using SGX with a persistent database incurs substantial overhead under specific conditions. Notably, experiments 410 and 510 induce additional latency attributed to intensive I/O operations. Specifically, each query in these tests repeatedly calls the libc function `fstat` to get the database file size, an operation further slowed by enclave OCALLs. We note that caching this information at the WASI layer could improve the performance of these recurring calls, provided that the file remains unmodified. Compared to the 410 test on an in-memory database, TWINE and SGX-LKL reveal slowdowns of up to 32.5× and 22.0×, respectively. Test 210 is I/O-intensive, as it modifies the database schema and, consequently, all records. Moreover, test 250 is highly I/O-intensive within a persisted database, as it updates every record in a table, requiring the re-encryption of a significant portion of the database file. Finally, 990 is a particular case of database housekeeping. It gathers statistics about tables and indices, storing the collected information in internal database tables where the query optimiser can access and use the information to help make better query planning decisions. The protracted execution time of TWINE and SGX-LKL with a persistent database is attributed to the added complexity of I/O operations from the enclave and the transparent encryption and decryption of the data. In general, a consistent pattern is observed across variants, except in cases involving extensive file system interactions, which typically influence native execution outside SGX as well.

6.6.4.1 SQLite: overhead breakdown

In order to comprehensively examine the origins of observed performance penalties, we devised a test suite to assess prevalent database queries. This suite comprises various query types, including insertion, sequential, and random reading (measured independently due to their dis-

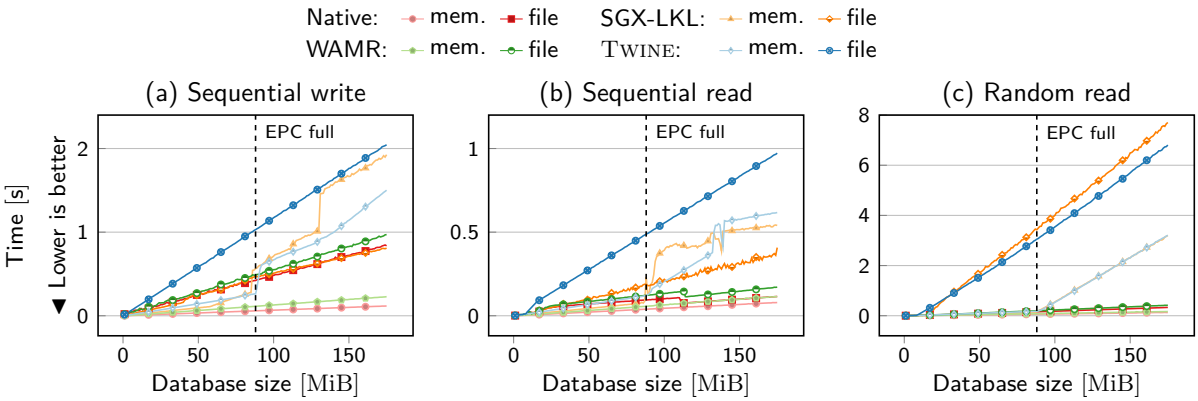


Figure 6.8: SQLite, micro-benchmarks for sequential `write` (a), sequential `read` (b), random `read` (c). We compare four variants (native, SGX-LKL, WAMR, TWINE) and two different modes (in memory and on file).

tinct complexities [62]). This test suite's design follows comparable benchmarks from previous literature [304]. The tests use a single table with an auto-incrementing primary key and a blob column. For sequential insertions, the blob column is iteratively filled by an array of random data (1 KiB) using the same pseudorandom number generator (PRNG), as in `Speedtest1`. Next, records are selected in the order they have been inserted (`WHERE` clause). Finally, we selected one random entry at a time. The database is initialised with 1 k records (*i.e.*, 1 MiB in total) and iteratively increases that amount by 1 k entries at the time, up to 175 k records (*i.e.*, 175 MiB). We evaluated four variants: a native version of SQLite running either outside or inside an enclave and an AOT-compiled Wasm version running with the same settings. For each of them, we include results for in-memory and on-file databases. The performance results for TWINE (file-based) illustrate the enhanced version of IPFS, which reduces the latency of the read/write operations. The details of the improvement of IPFS are covered in Section 6.6.4.3. Table 6.1 summaries the obtained results, where values on each line are normalised with the run time of the native variant. The run time is the median of the queries' execution time, either from 1 k to 175 k records for native and WAMR, or split into two parts for SGX-LKL and TWINE, going from 1 k to the EPC size limit (88 MiB of data and 4 MiB of runtime) and from that limit to 175 k.

Figure 6.8a presents the results of the insertion of records. While the variants executed outside the enclave demonstrate consistent performance, the EPC limitations impact the in-memory variants. This outcome can be attributed to the costly nature of swapping operations associated with enclave memory pages, which involves the encryption of evicted pages [305]. Importantly, TWINE demonstrates a significant performance improvement in the in-memory insertion above the EPC limit, with a gain of 1.3×. This improvement highlights the effectiveness of TWINE in addressing the performance challenges associated with EPC constraints. The operational cost associated with the persistent database in TWINE escalates linearly because of file encryption. Conversely, SGX-LKL employs a more optimal approach for inserting sequential elements and adheres to the performance trend exhibited by TWINE's in-memory variant.

Figure 6.8b illustrates the execution time required to read all records sequentially. Non-enclave variants exhibit linear costs, with a minor decline when the database accommodates 114 k records. Although this study focused on TWINE's performance, the slightly unexpected be-

	WAMR	SGX-LKL		TWINE	
		<EPC	≥EPC	<EPC	≥EPC
Insert memory	1.9	2.9	16.3	4.3	10.4★
Insert file	1.1	0.9	1.0	2.3	2.4
Sequential read memory	1.4	2.0	7.7	3.1	6.3
Sequential read file	1.3	1.3	3.1	3.8	8.3
Random read memory	1.3	2.3	17.6	2.9	17.6
Random read file	1.3	20.4	22.7	18.3★	20.1★

(Native run time = 1) ★TWINE is faster than SGX-LKL.

Table 6.1: The comparison of the technologies in normalised run time.

haviour has yet to be further investigated. Both TWINE and SGX-LKL with an in-memory database display a pronounced increase beyond the EPC size limit, attributable to enclave paging. In contrast, TWINE with a file-based database demonstrates optimal performance while the database remains within the 8 MiB range (*i.e.*, the configured cache size for SQLite). A similar increase is observed up to 16 MiB (twice the cache size). To confirm that this overhead is related to the cache, the cache size was increased to 16 MiB, revealing that the sharp increase ceases at 32 MiB. We observe comparable trends by replacing our WASI implementation by the one of WAMR, which uses direct POSIX calls without encryption. As a result, the primary source of the observed performance penalties is attributed to SGX memory access operations.

Figure 6.8c presents the execution time associated with random reading operations. The costs for all variants increase linearly in proportion to the database size, except for the SGX in-memory database variants, which are affected by EPC limitations. Random reading operations trigger the enclave paging mechanism more frequently, as the spatial locality of the requested records no longer remains smaller than the size of a single memory page. Notably, the in-file random reading scenario underscores the superior performance of TWINE compared to SGX-LKL, exhibiting a 1.1× increase prior to the EPC limit and a 1.13× increase following the limit.

As a result, TWINE exhibits somewhat lower performance compared to SGX-LKL, which can be attributed to the overhead associated with Wasm. Nevertheless, TWINE achieves competitive and even faster operations than SGX-LKL when it comes to random file access and in-memory insertion upon reaching the EPC threshold. While TWINE faces challenges in certain use cases, its overall performance remains robust. A detailed examination of the role of SGX in this behaviour is provided in the following section, which analyses the various cost factors of TWINE.

6.6.4.2 Cost factors assessment

To evaluate SQLite's performance limitations and constraints, we conducted a thorough analysis of the cost factors when using SGX and Wasm, either independently or conjointly. We identified two distinct dimensions of cost implications: (i) the time required to build and deploy an application, which occurs within the developers' workspace, and (ii) the time and storage space required to execute an application on an untrusted platform. The analysis presented herein was executed on the preliminary version of TWINE (before integration into the official repository).

(a) Times [ms]	Native	SGX-LKL	WAMR	TWINE
Compile runtime	—	288,774	4,329	3,425
Compile Wasm	—	—	38,593	38,593
Compile x86/AoT	23,350	23,350	52,944	52,944
Gen. disk image	—	15,711	—	—
Launch	2	6,119	70	3,155
(b) Sizes [KiB]	Native	SGX-LKL	WAMR	TWINE
Executable, disk	1,164	6,546	123	30
Enclave, disk	—	79,200	—	567
Wasm artifact, disk	—	—	1,155	1,155
AoT artifact, disk	—	—	3,707	3,707
Disk image	—	247,552	—	—
Executable, mem.	192,822	77,310	211,156	9,970
Enclave, mem.	—	261,120	—	209,920

Table 6.2: The cost factors of the micro-benchmarks.

Table 6.2a summarises the time overheads we observed with the SQLite micro-benchmarks (175 k records). As different types of costs are involved depending on the variant, we do not indicate totals in the table. The native one is composed of a single executable binary, while SGX-LKL requires the same executable binary and a disk image, which is an abstraction introduced to store the code and data securely. The two variants that use Wasm require an executable (the runtime) and the Wasm-compiled application corresponding to the SQLite benchmarks. For both variants, we measured the time for AOT compilation as well. For launching, we measured the time from the process creation to the start of the database initialisation. The variants without SGX are naturally faster since they do not have to initialise the enclave. The initialisation of TWINE is 1.9× faster than SGX-LKL because the enclave is heavier than TWINE's, and the benchmarks executable is encrypted on the disk image.

Table 6.2b indicates the disk and resident memory footprints of the compiled components and additional prerequisite software. The native variant is stored in a single executable binary file. SGX-LKL has a heavier-sized executable and a much larger enclave binary. The latter contains a generic program that is only loaded once and runs any other program stored in a disk image, such as our SQLite benchmarks. A disk image is necessary for SGX-LKL, which it maps into RAM. We generated an ext4-formatted file system, whose size is fixed at build time to be big enough to store our SQLite benchmarks programs and results. TWINE has a lightweight runtime, with a reduced memory footprint in the enclave, since the executable binary loaded into the enclave is only SQLite and the benchmarks. Besides, TWINE does not need an image file as it relies on the host file system, keeping its content secure thanks to IPFS. When loaded in RAM (last lines in Table 6.2b), the variants occupy different amounts of memory. Native and Wasm variants store the database records in the process address space (no enclaves). TWINE and SGX-LKL store records inside their enclaves, consuming less memory outside. The enclave sizes were configured to be just big enough to store 175 k records.

Finally, Figure 6.9 depicts the overhead incurred by the introduction of SGX in the breakdown of the micro-benchmarks using an in-file database. In particular, it compares the SGX hardware

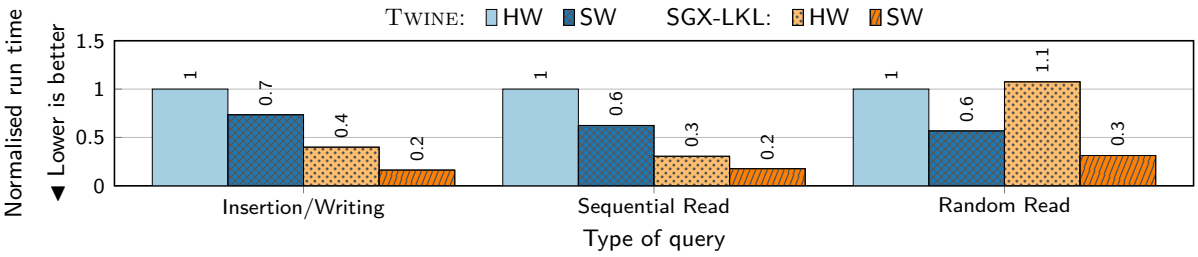


Figure 6.9: The comparison of normalised run time for SGX variants using in-file databases with SGX hardware and emulation.

mode, where the SGX memory protection is enabled and the software mode, where the SGX protection is emulated. The normalised run time is the median of the queries' execution time, from 1 k to 175 k records compared to TWINE in hardware mode. While the insertion and sequential reading time follow a similar trend, the performance of SGX-LKL in hardware mode for the random reading suffers from a slowdown. Since SGX-LKL in software mode does not encounter this issue, the performance loss is assignable to Intel SGX.

6.6.4.3 SQLite profiling and optimised IPFS

We conducted an in-depth profiling of SQLite primitives, with particular emphasis on examining the overheads from IPFS, wherein we observed the most significant slowdowns. The findings of this analysis have led us to the proposal of minor changes to the SGX SDK, which, remarkably, speed up the handling of protected files by a factor of 4.1×. Analogous to the previous analysis, we executed this profiling on the preliminary version of TWINE (prior to upstreaming).

We instrumented and profiled IPFS, which comprises two modules: one is statically linked with the enclave's trusted code, and the other is statically linked with the untrusted binary that starts the enclave. We broke down these two modules into distinct components, such as cryptography, node management, and API for trusted and untrusted environments. Additionally, manual instrumentation of the Wasm runtime was undertaken to profile each implemented WASI function associated with the file system. Profiling results exclude the enclave's execution time for current time retrieval (average 4 ms), as its repeated usage could lead to unexpected outcomes.

Time allocation for IPFS operations We identified the primary performance contributors for random reading as follows: (i) erasing memory (`memset`), (ii) invoking untrusted functions in the SGX SDK using OCALLs and calling POSIX functions, (iii) reading the database entries, and (iv) internal SQLite operations (*i.e.*, cache management). Figure 6.10 shows the costs of such operations while randomly reading the records. Within the IPFS bar of the breakdown plot, it is evident that 50.1% of the time is allocated to clearing the enclave's memory, 36.2% to transitioning between trusted and untrusted environments (for the retrieval of the file's content), 10.7% for reading operation, and a mere 2.9% dedicated to SQLite functionality.

Data storage in IPFS Internally, IPFS manages the protected file content by partitioning it into *nodes*, each corresponding to an encryptable or decryptable data block. These nodes are stored

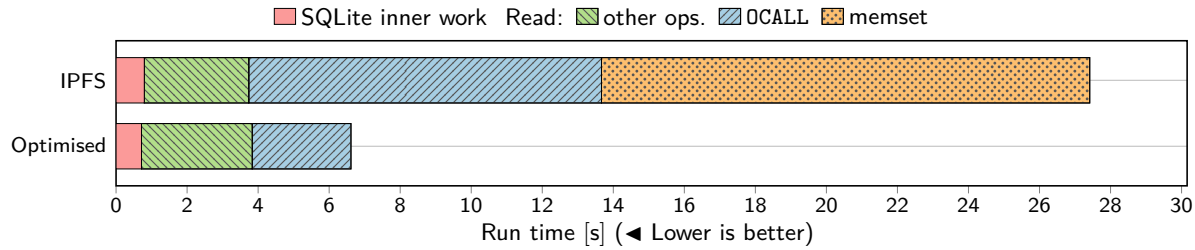


Figure 6.10: The run time breakdown prior to and following the optimisations of IPFS.

in a least recently used (LRU) cache, with each node containing two 4 KiB buffers for ciphertext and plaintext. Upon a node is added to the cache, the entire data structure containing its metadata is cleared. Given that an SGX memory page is 4 KiB in size [163], a minimum of two pages must be cleared, in addition to the metadata encompassed within the structure, such as node identifiers and various flags. Conversely, when a node is removed from the cache, the plaintext buffer is erased as well, corresponding to at least one SGX memory page.

Optimising IPFS Initialising structure data members in C++ prevents indeterminate values but may significantly impact performance in SGX enclaves. When adding nodes, functions set specific fields before clearing the node structure, then transfer the ciphertext into a designated buffer of the enclave memory, which is later decrypted into a corresponding plaintext buffer. Given this workflow, the sole requirement for initialising class data members involves assigning default values to the unassigned fields. We propose eliminating the clearing operations while assigning the remaining fields to zero. Thus, we preserve the original behaviour of the code, while sparing the valuable time to clear the memory of the structure, which is overwritten in any case. Similarly, upon a node is dropped from the cache, the plaintext buffer is cleared before releasing the node (*i.e.*, using C++'s `delete`). Although this practice is beneficial for purging the memory of confidential values when no longer needed, SGX provides protection for the enclave's memory. Given our threat model, no adversary can access this information, even if sensitive values remain in the SGX memory pages. For this reason, we also propose to remove the clearing operation for the plaintext in the discarded nodes.

Finally, we examine the time spent on reading the file content. The function responsible for this task issues an OCALL, crossing the secure enclave boundary to read the content of the database file. Our profiling measurements reveal that although untrusted POSIX calls are fast, a bottleneck exists in the code generated by the SGX tool `edger8r`, which interfaces the untrusted application with the enclave. The `edger8r` tool eases enclave development by generating edge routines to interface the untrusted application and the enclave and simplifying the process of issuing ECALLs and OCALLs. The edge functions responsible for reading files outside the enclave specify that the buffer containing the data must be copied from the untrusted application into the enclave's secure memory. IPFS decrypts the data after the OCALL and stores the plaintext into a buffer of the node structure, with 75.9% of the time is dedicated to completing this ciphertext copy. We propose eliminating this copy to the enclave altogether, opting instead to supply a pointer to the untrusted memory buffer to the enclave, from which the library can directly decrypt. In this revised implementation, adversaries may attempt timing attacks to

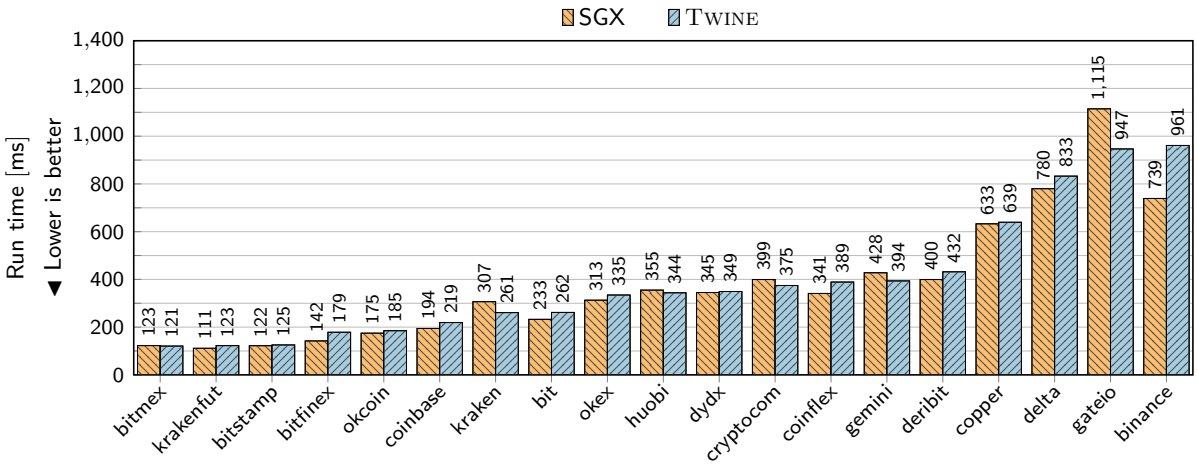


Figure 6.11: The pulling time of *Credora Private Pullers* using SGX and TWINE.

alter the ciphertext between data authentication and decryption, as the authenticated mode of operation for AES-GCM is *encrypt-then-MAC*. We suggest using AES-CCM, a *MAC-then-encrypt* cipher included in Intel's SGX SDK cryptography libraries. With AES-CCM, the authentication is verified based on decrypted data that is stored within the enclave secure memory. The cost of decrypting a block that fails authentication is minimal compared to a systematic buffer copy. Moreover, it remains an infrequent event in legitimate use cases.

The performance gains of our optimised IPFS can be seen in Figure 6.10 for random reading queries with 175 k records. The time previously devoted to clearing the memory has now been entirely eliminated, and the file reading operations now account for only 24.1% of the initial execution time. Compared to Intel's version, insertion achieves a 1.5× speedup and 2.5× for sequential reading. Ultimately, we have achieved a 4.1× speedup for random reading.

6.6.5 Macro-benchmarks: Twine for credit scoring

We conclude our experimental evaluation by estimating the impact of TWINE on the credit scoring application from *Credora*, as previously detailed in Section 6.5. Our goal is to evaluate the overhead of TWINE for time-sensitive functionalities impacting business operations. In particular, we focused on the *Private Puller* and *Aggregator* components, measuring the duration required to pull clients' data from Exchanges and to compute portfolio aggregations, respectively. In both instances, performance is critical, as producing outdated data may impair data accuracy and subsequently undermine *Credora*'s credibility.

The initial experiment measured the time to complete each request from single-thread clients. We target a variety of exchange venues (*e.g.*, Binance) by querying their API endpoints which return JSON data with a maximum size of 8 KiB, containing clients' positions (*i.e.*, an individual's ownership in a financial asset, reflecting their investment and potential for gains or losses). The baseline relies on an extended version of the `cpp-httplib` library [306], which uses SGX-WolfSSL [307] to carry out TLS cryptographic functions, adopted by *Credora* in production.

On the other hand, TWINE uses WolfSSL and Mongoose. Figure 6.11 reports the results of the *Private Puller* evaluation. It is apparent that the two solutions exhibit comparable performance results. For some exchanges, TWINE is slower (*e.g.*, up to 30% for Binance), while for others, it yielded better pulling time (*e.g.*, up to 17% for GateIO). This suggests that the performance is similar and subject to variations attributable to the server-side processing.

We measured the duration of a single aggregation round for all 522 client entries in the *Credora* production environment as of April 2023. Using nine parallel aggregators, the results were:

$$t(agg)_{no\_\text{TWINE}} = 3\text{m}23\text{s}$$
$$t(agg)_{\text{TWINE}} = 4\text{m}51\text{s}$$

The TWINE runtime overhead is acceptable for *Credora*, which can scale computing units, albeit at the expense of increased infrastructure costs. Profiling revealed the primary source of delay is attributable to the JSON library managing large data chunks. *Credora* uses the *nlohmann* library [308], which consumes substantial memory and performs suboptimally in TWINE. Future work will explore better memory-optimised JSON libraries to further mitigate this overhead.

6.7 Security analysis

This section analyses how TWINE contributes to the security requirements in Section 6.3.1 and compares its security posture to state-of-the-art solutions, namely AccTEE and SGX-LKL.

Security of $\mathbf{R}_1$ The two-way sandbox in TWINE offers security through two mechanisms: (i) Intel SGX, which protects against the system tampering with TWINE and the Wasm application, and (ii) , a Wasm sandbox to enforce memory safety while requiring the hosted application to rely on WASI for any interactions with the untrusted OS. These dual mechanisms allow both the application and infrastructure providers to confirm the integrity of the sandbox, which neither party cannot alter. For comparison, AccTEE also offers a two-way sandbox but lacks WASI support, thereby limiting the application's ability to access OS services and constraining the portability of legacy applications. SGX-LKL, conversely, does not offer a two-way sandbox but enables the execution of legacy applications through its ad-hoc variant of libc.

We observe that AOT-compiled code can bypass the Wasm sandbox if not compiled in a secure environment. Typically, the compilation of Wasm bytecode into assembly code ensures that memory access and function calls stay confined to the sandbox. However, malicious actors could craft assembly code that executes unauthorised operations in TWINE, like directly invoking an OCALL function. To counter this, we suggest three mitigations: (i) use JIT compilation to ensure secure code compilation within TWINE, albeit at a performance cost, (ii) establish a separate, secure enclave solely for compilation that communicates with TWINE, or (iii) coordinate with the Wasm application and infrastructure owners to validate the hash of the AOT-compiled assembly against the loaded code in TWINE, provided that both parties having prior knowledge of the Wasm bytecode. The first option is straightforward and already supported in TWINE (as it is based on WAMR), while the latter two are more challenging to tackle.

Security of R_2 and R_4 In TWINE, hosted applications can leverage pre-compiled Wasm components for secure communication, including a lightweight TLS library and an HTTPS library, enabling the creation of TLS-termination endpoints within enclaves to ensure data confidentiality and integrity during transmission. Furthermore, TWINE may extend Wasm's capability-based security model to permit application-level protocols exclusively like HTTPS, mitigating the data exfiltration risks through insecure channels. In contrast, SGX-LKL provides encrypted channels only between trusted enclaves or parties using Wireguard as the VPN solution. However, this approach has limitations: both endpoints must be configured with Wireguard, and including the TCP/IP stack in the TCB contributes to its increased size. Outside this trusted network, hosted applications must provide their own TLS implementation. SGX-LKL's oblivious communication feature is likewise confined to its VPN network. On the other hand, AccTEE claims to support I/O calls but relies on the application or underlying layer (*i.e.*, SGX-LKL) for encryption.

Security of R_3 and R_4 IPFS is integrated with WASI for secure file operations in hosted applications using transparent encryption and decryption. Replacing standard POSIX calls with IPFS functions mitigates the risk of exfiltrating sensitive data through the file system. The enclave's sealing key serves as the basis for a symmetric key, decrypting a file's Merkle tree root node. This root node holds essential metadata for decrypting subsequent nodes [298]. Although the Merkle tree nodes are stored on the untrusted file system, decryption is limited to the specific enclave or its owner via the enclave- or owner-bound sealing key [309]. SGX-LKL also offers file system security through the use of virtual block devices. It creates virtual disks on an untrusted file system and uses an ad-hoc algorithm to mitigate side-channel data leaks. While TWINE does not guard against side-channel attacks, its abstraction through WASI allows the secure file system to be replaced with other state-of-the-art solutions like Obliviate [69]. As for AccTEE, file system security is delegated to SGX-LKL, similar to its approach to network security.

Security of R_5 TWINE attests the Wasm bytecode using JIT compilation or the assembly code using AOT compilation, along with the runtime and SGX's TCB. Attestation serves two purposes: (i) it lets the enclave owner validate the genuineness of the hardware and the integrity of the application, and (ii) it provides assurance to the infrastructure owner of the correct implementation of the TWINE runtime. Furthermore, remote attestation ensures that a specific configuration of TWINE is in place, including disabling some system calls for the Wasm module. Turning off specific system calls enhances security by reducing the attack surface, adhering to the principle of least privilege, and easing system monitoring. By comparison, AccTEE mentions attestation but lacks an API to expose evidence to the Wasm applications for secure communication. SGX-LKL offers remote attestation by sharing the hash of the virtual disks with trusted entities. However, this approach can be challenging, especially when hosted applications store files since the attestation measurement also reflects these files. Consequently, managing multiple virtual disks becomes necessary for maintaining known attestation measurements.

6.8 Synthesis and next steps

Outsourcing computations to remote parties in distributed architectures raises trust concerns for sensitive applications. Although this issue has been extensively studied in the context of TEEs, such solutions introduce non-trivial drawbacks and constraints, including limited programming language support, restrictions on system calls, and mandatory programming paradigms. In this chapter, we propose an approach for executing unmodified programs in WebAssembly (Wasm)—a target binary format for applications written in LLVM-supported languages, such as C, C++, Rust, and Go—within a process-based TEE that can be easily deployed across client, edge computers and servers. TWINE is our trusted runtime that supports the execution of unmodified Wasm binaries within SGX enclaves. Wasm offers several advantages, including speed, versatility, and abstraction of the complexity associated with developing applications tailored for specific TEEs. Furthermore, we provide an adaptation layer between the standard WebAssembly System Interface (WASI) used by applications and the underlying OS, translating WASI operations into equivalent native system calls or functions from secure libraries specifically designed for SGX enclaves. Consequently, trusted applications can transparently interact with encrypted files and secure network connections via TLS and HTTPS. Our comprehensive evaluation demonstrates performance comparable to other state-of-the-art approaches while offering robust security guarantees and full compatibility with standard Wasm applications. TWINE is freely available as open-source software and has been merged into the original WAMR runtime.

Bringing secure computations on resource-constrained IoT devices is the next challenge addressed in this dissertation. These peripherals typically operate in constrained environments with limited processing power and memory capacity. Some of these devices are built with Arm TrustZone, a partition-based TEE allowing manufacturers to establish trust. The following chapter explores, designs, and implements a trusted Wasm runtime for secure execution by leveraging this TEE implementation, aiming to promote Wasm across the cloud-edge continuum.

Chapter 7

A trusted WebAssembly runtime for Arm TrustZone: WaTZ

The previous chapter presented WebAssembly (Wasm) as a trusted technology for secure software execution using Intel SGX and designed for server-grade machines and edge computing. This chapter extends Wasm support to resource-constrained IoT devices using Arm TrustZone, through the introduction of WaTZ, a trusted Wasm runtime built on top of the trusted OS OP-TEE. Although TrustZone lacks native support for attestation, WaTZ leverages a root of trust to create a secure environment for trustworthy software execution. Using this approach, WaTZ provides Wasm applications with an API for remote attestation, offering security guarantees comparable to those of Intel SGX. This work further illustrates how Wasm can be used as a portable and secure solution for confidential computing, spanning the cloud-edge continuum.

Chapter outline

7.1 Introduction . . . 112
7.2 Related work for WaTZ . . . 113
7.3 WaTZ: system overview . . . 114
7.3.1 Threat model . . . 114
7.3.2 Design overview . . . 115
7.3.3 Embedded trusted runtime . . . 116
7.4 Remote attestation of WebAssembly . . . 117
7.4.1 WaTZ protocol for remote attestation . . . 118
7.5 Implementation . . . 120
7.5.1 The runtime (attester) . . . 121
7.5.2 The server (verifier) . . . 122
7.5.3 The attestation service . . . 122
7.5.4 Extension to WASI: WASI-RA . . . 123
7.6 Evaluation . . . 124
7.6.1 Time measurements in TrustZone . . . 125
7.6.2 Startup overhead . . . 125
7.6.3 Micro-benchmarks: PolyBench/C . . . 127
7.6.4 Macro-benchmarks: SQLite . . . 128
7.6.5 Micro-benchmarks: remote attestation . . . 129
7.6.6 Macro-benchmarks: remote attestation with Genann . . . 131
7.7 Security analysis . . . 133
7.8 Synthesis and next steps . . . 134

7.1 Introduction

Security is paramount in designing and deploying distributed applications for mutually distrusting stakeholders, such as infrastructure providers, software developers, and data owners. The complexity of the problem increases in heterogeneous systems when considering decentralised operations of IoT, edge, and cloud devices, which are all at risk of being compromised.

Trusted execution environments (TEEs), such as Intel SGX, TDX [102, 107], Arm TrustZone, CCA [104, 108], and AMD SEV/-ES/-SNP [182, 184, 185] and RISC-V TEEs [103, 105], offer hardware support for securely executing applications in shielded environments. While these isolated environments are promoted by the commercial offerings of major cloud providers, they do not guarantee that the code itself is trustworthy and has not been tampered with. Remote attestation is typically used to assess the code before its execution. However, while this key security feature is provided by some TEEs (*e.g.*, Intel SGX), Arm TrustZone lacks built-in support for remote attestation. This is concerning given the increasing adoption of Arm-based architectures in IoT edge computing [310] and general computing market [311]. In addition, recent attacks have compromised IoT devices via malicious firmware updates [312] or software flaws [313], often beyond TrustZone's threat model. The inability to verify the authenticity of software upon execution renders these platforms inadequate for handling sensitive tasks and data, particularly in recent scenarios such as trustworthy machine learning systems at the edge [314].

Wasm, a recent portable compilation target and bytecode standard, enables building applications using modern programming languages with near-native execution speed, while supporting legacy code since modern compilers support Wasm [240]. TrustZone is a constrained environment that runs small executables using a specialised API. Wasm is well-suited in this model, thanks to its small runtime overhead, portability, supporting non-standard system interfaces, and fast as the bytecode can be just-in-time (JIT) and ahead-of-time (AOT) compiled.

Embedding Wasm in Arm TrustZone has many potential applications. The smartphone industry can use Wasm as an interoperable bytecode for executing secure applications within TrustZone, currently limited to manufacturers' or strategic partners' needs [315]. In the automotive sector, IoT devices could leverage Wasm to run machine learning algorithms inside enclaves, ensuring the validity of the results thanks to attestation. Section 7.6.6 demonstrates this use case.

This chapter presents WATZ, an efficient and secure runtime for trusted execution of Wasm code inside Arm TrustZone, with added support for remote attestation. The sandbox isolation of Wasm is leveraged to mitigate and potentially prevent vertical privilege escalations and lateral attacks. We combine TrustZone with Wasm bytecode to issue trustworthy evidence attesting to the genuineness of running software. In Section 7.4, we adapted and fully implemented the remote attestation protocol inspired by SGX EPID (covered in Section 3.4.2), using a public-key infrastructure. As such, we facilitate the deployment of fully decentralised applications spanning various devices at the core or the edge of the network. WATZ extends OP-TEE [111], a popular open-source trusted OS, and we validated our prototype with Arm hardware.

The main contributions presented in this chapter are summarised as follows: (i) WATZ: the first system to run Wasm applications in TrustZone, leveraging WebAssembly System Interface (WASI) to interact with the TEE's trusted API and the untrusted OS. WATZ offers trans-

parent communication with TrustZone for hosted Wasm applications and enhanced isolation, allowing each trusted application (TA) to be fully isolated from others and the trusted OS, similar to SGX enclaves (Section 7.3); (ii) A remote attestation protocol for Wasm in TrustZone (Section 7.4), ensuring trustworthy execution in the absence of built-in attestation. WASI-RA, an extension of WASI, enables hosted Wasm applications to attest against trusted parties and securely communicate confidential data, based on Intel SGX's remote attestation protocol. WaTZ provides an end-to-end trust solution leveraging the fingerprint of isolated Wasm applications, provided that some common hardware features are available (*e.g.*, root of trust, secure boot); (iii) A Comprehensive evaluation (Section 7.6) showing WaTZ's performance on par with Wasm execution outside TEEs and up to 2.12× compared to native execution, deemed negligible given the proposal's benefits. Contributions to the trusted OS (OP-TEE) enable AOT compiled Wasm applications to achieve these results. A security analysis of WaTZ and formal verification of the attestation protocol (Section 7.7) complement the evaluation.

7.2 Related work for WaTZ

Building upon the background part presented earlier in this thesis (§I), we survey more specialised related work for WaTZ, focusing on TEEs and attestation for devices with TrustZone.

WebAssembly and TEEs Few options exist to host Wasm applications inside TEEs. The previous chapter introduced Twine, an trusted runtime for executing Wasm applications inside Intel SGX enclaves. Twine relies on the WebAssembly micro runtime (WAMR) [117] with WASI for secure file system interactions. Enarx [283] targets Intel SGX enclaves and AMD SEV virtual machines. Veracruz [284] only supports VM-based TEEs, such as Arm CCA [108] and AWS Nitro [285] enclaves, having dropped SGX and TrustZone enclaves considered too constraining [316]. AccTEE [120] and Se-Lambda [281] run Wasm binaries using Intel SGX using the V8 JavaScript/Wasm engine. AccTEE provides trusted resource accounting, while Se-Lambda deploys serverless programs over function as a service (FaaS). Contrary to the mentioned solutions, WaTZ runs on Arm TrustZone, uses WASI for system interactions and supports trustworthy code execution, and supports trustworthy execution via close Wasm integration and attestation. We propose an enhanced paradigm for Arm TrustZone: every hosted Wasm application is isolated from the rest of the trusted world via the Wasm sandbox. We note that OP-TEE requires every TA to be signed to be trusted and executable in the trusted world. This is a significant impediment when offering trusted execution for third parties, which WaTZ addresses. This approach maintains the security of TrustZone thanks to the isolation of Wasm sandbox.

Table 7.1 compares WaTZ against state-of-the-art trusted TEE development frameworks for Wasm-compiled applications, along some key characteristics which are notable for IoT devices. Each feature can either be missing (○), or fully (●) available.

Remote attestation Recent work tackles remote attestation mechanisms and protocols for IoT and edge devices [103, 127, 223, 317–319]. Intel SGX has built-in support for attestation [122]. WaTZ follows similar principles to deliver attestation on Arm processors. Ling *et al*. [320] proposed a trusted boot mechanism with remote attestation of software running in the

Features	Twine	Veracruz	Enarx	AccTEE	Se-Lambda	Teacleave	WaTZ
ahead-of-time (AOT) compilation	●	○	○	○	○	○	●
WASI support	●	●	●	○	○	○	●
Remote attestation	●	●	●	○	●	●	●
Using attestation with WASI	○	○	○	○	○	○	●
Micro runtime (< 1 MiB memory)	●	○	○	○	○	●	●
Target TEE for IoT devices	○	○	○	○	○	○	●
Supported TEEs	SGX	CCA, Nitro	SGX, SEV	SGX	SGX	SGX	TrustZone
Programming language(s)	C	Rust	Rust	C++, JavaScript	C++, Go	Rust	C

Table 7.1: The comparison of the related work features.

normal word of Arm TrustZone. Similarly to WaTZ, they leveraged OP-TEE to ensure IoT devices' boot and runtime state trustworthiness. However, their approach depends on the integrity of the normal world kernel. In contrast, WaTZ runs attested software isolated in the secure world, without depending on a reliable normal world OS. WaTZ extends the design of SGX EPID attestation protocol (covered in Section 3.4.2) to use software enclaves created in the trusted world, through isolation within Wasm sandbox. Strongly integrated with Wasm via WASI-RA, WaTZ is the first system enabling hosted applications to control the remote attestation process by measuring the Wasm bytecode to issue evidence. As a result, WaTZ enables remote software appraisal and IoT edge device authentication by leveraging TrustZone, a hardware-based root of trust and secure boot to build a hardware-enforced attesting environment.

7.3 WaTZ: system overview

This section introduces the trusted runtime WaTZ, its threat model, how it is supposed to integrate with Arm's system-on-chips (SoCs) and the underlying trusted OS, its architecture, and the available execution modes for Wasm-compiled applications running in the secure world.

7.3.1 Threat model

We consider three aspects as a threat when designing and implementing WaTZ: (i) the hardware, which is the operating SoC or board, (ii) the secure world of TrustZone, which corresponds to the TEE, and (iii) the normal world of TrustZone, the part of the system outside the TEE, but cooperates with the secure world on regular basis to provide system services.

WaTZ leverages the following hardware features: (i) TrustZone security extensions, (ii) a root of trust, and (iii) secure boot. We assume an adversary with physical access to the device, but unable to subvert the hardware protections. As such, the adversary may fully control the SoC and its peripherals, but excluding all the components belonging to TrustZone. Consequently,

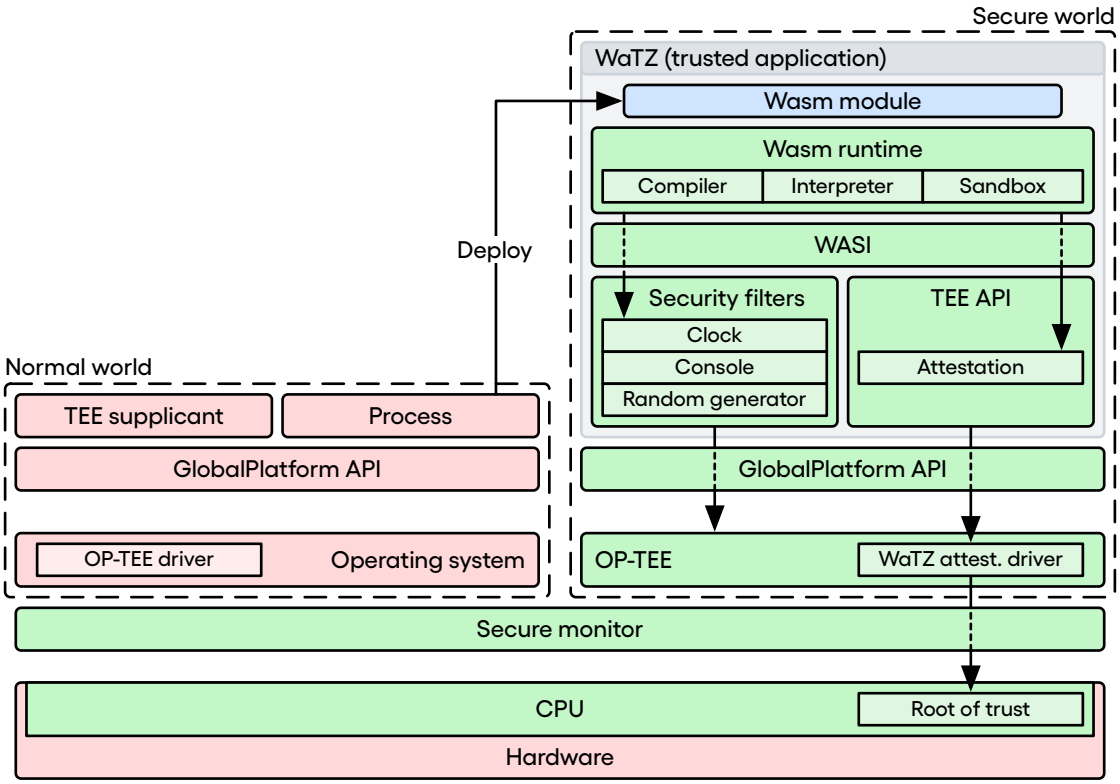


Figure 7.1: The architecture of WATZ. The green boxes (▢) are the TCB of Arm Trustzone. The red boxes (▢) are untrusted. The blue box (▢) is the Wasm module that is attested.

WATZ cannot defend itself from physical attacks to the volatile memory space assigned to TrustZone, which is not encrypted, unlike Intel SGX. We also consider storage rollback attacks out of scope, which can be mitigated using hardware monotonic counters [321].

Regarding the normal world, we assume the secure monitor, bootloader, and trusted OS do not contain vulnerabilities enabling an attacker to breach the TEE. The cryptographic primitives and algorithms are considered correct. Code and data inside the TEE are trusted and cannot be accessed from the normal world, except through dedicated channels controlled by WATZ. Finally, side-channel attacks [322–326] are out of scope, and mitigations exist [327, 328].

We make no assumption regarding the normal world, which includes the rich OS and the user space. Compromised OSes may arbitrarily respond to trusted OS calls, causing its malfunction. The TAs relying on the normal world services should be carefully crafted to ignore abnormal responses or even abort execution in such cases.

7.3.2 Design overview

WATZ is a trusted runtime to execute Wasm applications with a remote attestation mechanism. Figure 7.1 illustrates its components. Its small footprint (256 KiB on disk, including the runtime and WATZ's components) brings several advantages. Smaller programs offer smaller attack

surfaces. TEEs and small devices (edge or IoT) are usually tight in memory. Its small footprint allows deploying and running a complete Wasm virtual machine inside TrustZone using a small edge-scale Arm processor. We estimate that the increase of the trusted computing base (TCB) due to the embedding of the Wasm runtime in the TEE is outweighed by its benefits.

To provide trusted OS features to Wasm applications, we contribute an adaptation layer that binds WASI to the API existing in the trusted environment (the GlobalPlatform (GP) API in our prototype). This, in conjunction with the reliance of Wasm applications on WASI, allows WATZ to run unmodified Wasm applications, while benefiting from a TEE. While we support a subset of WASI enabling the execution of general-purpose software, the adaptation layer is effortlessly extensible to support additional system calls, such as file system and networking.

To offer remote attestation, we designed and implemented WASI-RA, an extension of WASI enabling the hosted Wasm applications to interact with the process of attestation. We guarantee that WATZ is booted correctly and was not tampered with using a secure boot system. When WATZ loads a Wasm application, its bytecode is stored in the secure memory and measured to produce a hash. Using our interface, a hosted application can request WATZ to generate trusted evidence based on the hardware secret and the hash of the Wasm bytecode. Further, WATZ includes a remote attestation protocol to enable a third-party verifier to check if evidence is genuine, which relies on the measured fingerprint of the Wasm applications. Upon positive attestation, our protocol simplifies the establishment of a hybrid cryptosystem for the verifier and the attester, which can be later used to create secure communication channels.

7.3.3 Embedded trusted runtime

We built WATZ as an embedded Wasm runtime with WASI. This design provides several advantages, *i.e.*, removing some barriers for building TAs. First, WATZ opens the choice for programming languages. Provided that the compiler can emit Wasm bytecode and supports WASI, it is a clear advantage over vanilla OP-TEE, which limits developers to C only. Second, this hides the complexity of writing code dedicated to OP-TEE since WASI calls are generated by the compiler, and they abstract the implementation details of GP API. Furthermore, Wasm separates the virtual address spaces used for Wasm applications and the native runtime process, and code, stacks and heap are handled separately, making memory-oriented attacks or developer mistakes more unlikely. Besides, WASI ensures that the applications do not harm the secure world and acts as a gatekeeper to run operations outside the runtime. Finally, applications are not tightly coupled to the underlying TEE, and WATZ can load any regular Wasm/WASI application without changes. As a result, Wasm brings more flexibility, versatility and security compared to its native counterpart, coupled with its strong sandboxing mechanism that isolates each hosted Wasm application. This newly introduced isolation layer extends the single trusted world of TrustZone to an environment with multiple secure and mutually distrusting enclaves, similar to the isolation scheme offered by Intel SGX. Hence, WATZ changes the paradigm of software deployment and execution of the secure world to a more relaxed approach, which previously required TAs to be signed, while not introducing any security drawback.

Instances of Wasm-compiled TAs are started by a user space process in the normal world, which uses the standard OP-TEE API to prepare a buffer containing the Wasm binary and calls WATZ in

the secure world. Once in the secure world, WATZ copies the bytecode into a secure, sandboxed memory, computes the hash for future attestation and starts the execution. Wasm applications use the WASI to interact with the OS, which, in turn, diverts the calls to WATZ. We implemented the WATZ attestation service as a new kernel module for OP-TEE, enabling the runtime to generate evidence for relying parties to prove the applications' authenticity. While several open-source runtimes exist to execute Wasm code, we settled for WAMR [117] (as explained in Section 4.4.3), a lightweight and embeddable runtime implemented in C, ideal for TEEs in general and OP-TEE in particular, since TAs are written in C.

WAMR, the underlying Wasm runtime we have chosen for WATZ, can execute code in three modes, each with its benefits and drawbacks: interpreted, just-in-time (JIT) and ahead-of-time (AOT) compiled. These three execution modes were previously described in Section 4.2.3. We opted for the interpreted mode and AOT compilation for WATZ's runtime, as JIT compilation requires embedding a compiler in the runtime, which would increase its complexity, size and dependencies. OP-TEE OS imposes a strict memory capacity for TAs, which impedes the usage of a JIT compiler in the trusted world (further elaborated in Section 7.5).

When using AOT compilation, the bytecode is translated before execution, so the runtime does not need to include a compiler but requires the TA to allocate executable memory after the process has started in the user space of the trusted world. However, OP-TEE's API lacks any memory management system calls for modifying the page attributes to mark them as executable [329]. Hence, we extended the trusted kernel to provide such functionalities to TAs. Using AOT compilation, the execution speed is, on average, 28× faster than with interpretation.

7.4 Remote attestation of WebAssembly

This section presents the remote attestation mechanism, first by explaining how the hardware is trusted, extending this principle to the secure OS, WATZ and the Wasm hosted applications. We designed WATZ for devices that expose a hardware *root of trust* to the secure world. We extended OP-TEE to deterministically derive a key pair from the hardware root of trust. Normal and secure OS can hence be updated without losing the key materials, and on-chip key generation guarantees that the private key never leaves the trusted kernel OS. The public key is then exported and used as an endorsement value to be verified during remote attestation requests. This key pair, called *attestation keys*, is at the core of WATZ's mechanisms to provide attestable signatures and guarantee platform authenticity.

Secure boot is a security mechanism to ensure that the device is booting in a trusted state. WATZ requires the device to implement secure boot, so only trusted actors are able to provision software to boot the secure world, *i.e.*, only signed bootloaders and OS can boot the SoC. Therefore, this enforces a *chain of trust* that protects the attestation keys against extraction from the secure kernel OS. The boot sequence is as follows: the first-stage bootloader (ROM) verifies if the second-stage bootloader is genuine, based on the public key stored in one-time programmable fuses (eFuses) [330]. The previous booting component recursively verifies the next boot stages until the secure world is fully booted. Afterwards, the normal world is started and exposes a daemon serving system services to WATZ, such as access to the file system.

WaTZ generates cryptographically signed reports, called *evidence*, asserting that an executing Wasm application is trustworthy and running on a genuine device, by producing a hash of the Wasm AOT binary stored in the secure memory at launch time, among other security information. We offer an API to Wasm-compiled applications (see WASI-RA in Section 7.5) to issue evidence and establish a secure communication channel with a verifier. Then, the evidence is checked by the verifier using the corresponding public attestation key of the device and examines the code measurement to match its reference values.

The evidence is created by interacting with the attestation service, implemented as a kernel module in OP-TEE (shown in Figure 7.1). The evidence includes (i) an anchor, which is a value defined by the transport layer to bind security parameters to a particular session (*e.g.*, a public session key), (ii) the version of WaTZ, enabling the relying party to exclude outdated systems, (iii) the claim, *i.e.*, the Wasm binary hash, (iv) the public key of the attestation service, for the verifier to determine if the device is endorsed, and (v) the digital signature of the evidence.

Security requirements WaTZ's remote attestation protocol satisfies a number of security properties and requirements, which are further elaborated as follows.

- $\mathbf{R}_1$. **Mutual key establishment**: A shared secret key is established for communication between the attester and the verifier, using the elliptic curve Diffie-Hellman ephemeral (ECDHE) key-agreement protocol. This secret is later used for symmetric encryption.
- $\mathbf{R}_2$. **Mutual entity authentication**: The attester and the verifier are mutually authenticated to prevent masquerading attacks. From the attester's standpoint, the verifier's public key must be hardcoded into the Wasm application. Hardcoded values combined with the application measurement ensure that an attacker cannot change the key without invalidating evidence, so the software can only communicate with the intended remote service.
- $\mathbf{R}_3$. **Half trust assurance**: The attester proves to the verifier that it is executing on a trustworthy platform and with the expected Wasm application. The verifier does not reciprocate to the attester, so the attester assumes the entity authentication is sufficient.
- $\mathbf{R}_4$. **Freshness**: ECDHE (*ephemeral*) key-agreement protocol requires attester's and verifier's key pair to be fresh, hence preventing replay attacks.
- $\mathbf{R}_5$. **Forward secrecy**: Compromised long-term secrets do not affect the security of earlier or future exchanges. Similarly to freshness, ECDHE key-agreement protocol achieves this goal, which means the keys are renewed for every tentative of remote attestation.

7.4.1 WaTZ protocol for remote attestation

The GP API defines an interface to establish a secure communication channel using TLS. However, OP-TEE lacks the corresponding implementation of this interface [331, 332]. We extended and implemented the remote attestation protocol of Intel SGX Enhanced Privacy ID (EPID) [149] (itself inspired by SIGMA [333], a family of key-exchange protocols) not to rely on TLS. We changed the protocol compared to the original in various aspects: (i) removed the SGX specificities, such as the interaction with the quoting enclave, as the kernel module of

Sender		Receiver	Message structure
Attester (A)	$\Longrightarrow$	Verifier (V)	$\text{msg}_0 := G_a$
Verifier (V)	$\Longrightarrow$	Attester (A)	$\text{msg}_1 := \text{content}_1 \,\|\|\, \text{MAC}_{K_m}(\text{content}_1)$
			$\hookrightarrow \text{content}_1 := G_v \,\|\|\, V \,\|\|\, \text{SIGN}_V(G_v \,\|\|\, G_a)$
Attester (A)	$\Longrightarrow$	Verifier (V)	$\text{msg}_2 := \text{content}_2 \,\|\|\, \text{MAC}_{K_m}(\text{content}_2)$
			$\hookrightarrow \text{content}_2 := G_a \,\|\|\, \text{evidence} \,\|\|\, \text{SIGN}_A(\text{evidence})$
			$\hookrightarrow \text{evidence} := (\text{anchor} \,\|\|\, A \,\|\|\, \ldots)$
			$\hookrightarrow \text{anchor} := \text{HASH}(G_a \,\|\|\, G_v)$
Verifier (V)	$\Longrightarrow$	Attester (A)	$\text{msg}_3 := iv \,\|\|\, \text{AES-GCM}_{K_e}(data)$

Table 7.2: The remote attestation protocol of WATZ.

WATZ provides the measurements, (ii) merged the two first messages to communicate from the client to the server as they tightly relate, (iii) provided a fixed structure for the last message to transparently handle confidential data, eliminating the burden of a hosted Wasm application from decrypting that content, (iv) omitted Intel's SGX EPID for conciseness, and (v) removed the dependency on Intel's public key infrastructure (PKI), since the device's key pair is emitted by WATZ based on the embedded root of trust. Table 7.2 formalises the remote attestation protocol. Below, we detail each protocol's message and the related cryptographic operations.

(a) Message 0 (attester → verifier): The attester generates a session key pair $\langle a, G_a \rangle$ and sends the public part G_a to the verifier.

(b) Message 1 (attester ← verifier): Upon reception of msg_0, the verifier generates a session key pair $\langle v, G_v \rangle$. It computes the shared secret from the public session key of the attester G_a and its private session key v, which gives G_{av}. This shared secret is derived into a key derivation key (KDK), which is further derived into two shared secrets: K_m for calculating message authentication codes (MACs) and K_e for future messages encryption in the session. These derivations are the same as in Intel SGX EPID [149]. The message msg_1 is replied to the attester, containing G_v, the verifier's elliptic curve digital signature algorithm (ECDSA) public key V (its identity), and a signature of both public session keys. The message is appended with a MAC.

(c) Message 2 (attester → verifier): The attester verifies the signature of the public session keys: different session keys may reveal a masquerading or replay attack, and assesses the MAC of msg_1. It also checks whether the service public key V matches the hardcoded key in the Wasm application. Doing so ensures the attester communicates with the intended service and prevents an attacker from altering that key as it is part of the code measurement. The attester computes the shared secret from the public session key of the verifier G_v and the private session key a, which gives G_{va} that is equal to G_{av} computed by the verifier. The key derivations follow the same process as in msg_1. The attester creates msg_2 by concatenating its public session key G_a with a newly generated evidence signed by the attester A, where the anchor of the transport

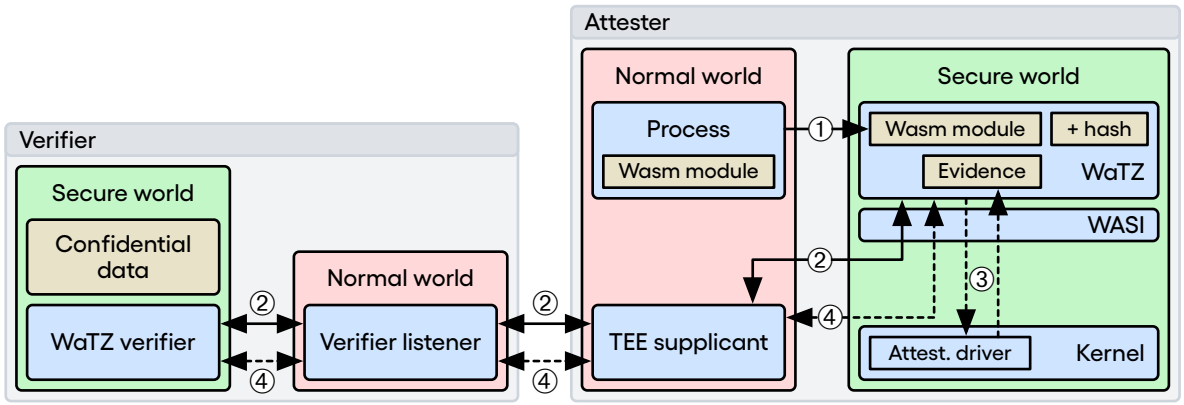


Figure 7.2: The architecture of the prototype. The solid arrows describe untrusted channels, whereas dashed arrows are trusted channels. The green boxes (▢) represent the secure worlds, the red boxes (▢) are the normal worlds, the blue boxes (▢) are software components, and the brown boxes (▢) are the data or code.

layer is the hashed concatenation of the public session keys. Finally, the attester authenticates msg_2 by appending a MAC.

(d) Message 3 (attester ← verifier): The verifier checks the MAC of msg_2 and verifies that G_a matches the one previously received in msg_0. It also examines whether the anchor corresponds to the public session keys, revealing a masquerading or replay attack. It extracts the evidence's public attestation key and checks against its list of endorsed public keys to determine whether this is a known device. If the key is found, the digital signature of the evidence is checked, which indicates whether the hardware is genuine. Finally, to verify that the Wasm application is trustworthy, its code measurement claim is compared with a list of possible reference values. If all verifications pass, the protocol sends msg_3 with an arbitrary confidential data, called *secret blob*, encrypted with AES-GCM, which requires the initialisation vector iv.

We simplified the protocol by omitting the use of session identifiers. Such identifiers are needed for having multiple sessions with concurrent remote attestation requests. We also reduced the complexity by keeping evidence in clear. If the secrecy of this structure is a concern, the protocol can be further enhanced to protect evidence using an encryption scheme like AES-GCM. Section 7.7 describes how Scyther, a formal analyser, has been used to verify the protocol.

7.5 Implementation

This section described the implementation details of WaTZ. Note that our prototype requires the following features from the underlying hardware platform: (i) a root of trust, (ii) a secure boot to harden a chain of trust, and (iii) the TrustZone extensions in the CPU. WaTZ relies on OP-TEE, and we detail the extensions and implementation details in the following.

Overview of WaTZ Figure 7.2 illustrates the components of our prototype, which reflects the architecture defined in Section 7.3. It comprises an attester and a verifier. Initially, a Wasm-

compiled application is loaded by the normal world in the TEE (①). WATZ runtime initialises the runtime using the Wasm binary, measures it, and executes the application. Later on, the hosted application can use our WASI extension for remote attestation to fetch the secret blob from a relying party. This consists of contacting a verifier to create a secure communication channel (②). An anchor value is associated with the remote attestation handshake, which is forwarded to the attestation service in the trusted kernel to issue evidence (③). Lastly, the verifier checks whether the device is genuine, based on the evidence. If deemed trustworthy, the verifier sends the secret blob using the secure channel (④). In the remainder of this section, we describe further details of the inner working of the attester, verifier and attestation service.

7.5.1 The runtime (attester)

Our Wasm runtime is a trusted application written in C (1.6 k SLOC), statically linked to a modified version of WAMR. As Wasm-compiled applications call the OS using WASI, we implemented an ad-hoc WASI layer, mapping the calls to the functions available in the GP API. To find all the required calls used to execute our experiments properly, we first manually coded dummy functions for all 45 WASI functions, throwing exceptions when called. Then, we implemented adapters for the WASI functions necessary for our benchmarks (0.9 k SLOC) to use whatever was available in the GP API. While our experiments could successfully be executed without adding extra features to GP, many standard functionalities (*e.g.*, thread management) are not supported by TAs. However, we note that with some extra engineering efforts, WATZ may be completed to support file system interaction via OP-TEE's trusted storage API and supports other missing functions leveraging (parts of) an embedded C library or a library kernel.

Network communication The GP API includes TCP/IP sockets, implemented by OP-TEE. Under the hood, the trusted kernel redirects the communication to the normal world, using a shared memory buffer to transfer data. OP-TEE comes with a built-in user space supplicant daemon that runs in the normal world and is responsible for providing OS services.

Memory constraints When using WATZ, the Wasm-compiled application is copied from the normal world into the TA of the Wasm runtime in the trusted world. The implementation of OP-TEE does not allow TAs to access memory from the normal world directly. Instead, the normal world allocates a shared memory buffer accessible by both worlds. OP-TEE limits the amount of memory available for shared buffers. We increased the limit to 9 MiB, which is the largest value that would not break OP-TEE. A similar problem occurs when allocating memory inside the trusted world, which we modified to allow up to 27 MiB. Pushing further the memory limits leads to OP-TEE malfunctions. A quick investigation of its code indicates the likely reasons to be in the data structures used to maintain memory regions, not the Arm architecture itself [332]. According to the GitHub repository, increasing the memory cap of OP-TEE is a recurring request, and workarounds vary depending on the hardware. We note that such limits could be pushed much further away with recent research [334]. As our implementation and experiments did not require substantial amounts of memory, we did not address these constraints.

AOT compilation Wasm AOT bytecode is loaded by the runtime rather than by the native process loader of OP-TEE. As such, WATZ is required to allocate executable memory pages to store the AOT-compiled bytecode. We extended OP-TEE to provide such a feature by implementing an additional system call similar to POSIX's `mprotect`. Consequently, WATZ can allocate executable memory pages to run AOT Wasm bytecode, which is ARM64 assembly code. We plan to submit a pull request with this improvement to the OP-TEE upstream repository.

7.5.2 The server (verifier)

Our prototype implements a server application that acts as a verifier in the remote attestation protocol. It comprises two parts: (i) the listener in the normal world, and (ii) the verifier as a TA in the secure world. While the verifier is currently implemented as a TA written in C (1.4 kB SLOC), any environment that supports the cryptographic operations mentioned in Section 7.4 can implement and host this service. As such, the remote attestation server could be executed using another type of TEE like Intel SGX, or as a plain service in a traditional OS.

Network communication Contrarily to the attester, the verifier must have a dedicated application in the untrusted side of the TEE because the GP API for sockets lacks the capability of listening for incoming connections. For that matter, the messages are received in the listener application and forwarded to the TA. Similarly, the messages generated by the TA are handed to the untrusted application for delivery. Messages are stored in shared memory buffers, and their handling occurs in the secure memory, acting as the TEE-supplicant provided by OP-TEE.

Memory constraints The listener invokes the verifier for every message that is received and sent. We noticed that repeatedly calling TA's functions with large buffers (>3 MiB) caused OP-TEE to raise an error and abort the execution of the TA. Following the same principle as for the attester, we limited our micro-benchmark that assesses the transfer of confidential data (Section 7.6.5) to a maximum of 3 MiB. We mitigate the problem by making OP-TEE reuse the same buffer across several TA function calls, but we left this improvement for future work.

7.5.3 The attestation service

We designed WATZ to offload the signing of evidence to a dedicated trusted kernel module, called the attestation service (0.5 k SLOC). It plays a critical role in WATZ as it has access to the private attestation key. The attestation service, located in the kernel space of OP-TEE, prevents the key materials from being exposed to the TAs in the user space, acting as a hardware security module (HSM). Hence, the Wasm runtime communicates claims to the attestation service for issuing evidence, which is returned as a payload to the hosted application.

Root of trust In order to establish a root of trust, our hardware is equipped with a Cryptographic Accelerator and Assurance Module (CAAM). The root of trust is a unique 256-bit one-time programmable master key (OTPMK), fused into hardware at manufacturing time. The CAAM provides two different hashes of OTPMK, depending on if the requesting thread is in the normal or in the secure world. This hash is called the master key verification blob (MKVB). The MKVB

is then used as a seed to provision secrets only known by a kernel module in OP-TEE [330, 335]. We modified OP-TEE to support the full MKVB size, as it was limited to 128-bit hardware keys.

CAAM early usage and lock-up The CAAM is disabled by default in OP-TEE, because of sharing issues between the normal and secure worlds. When the CAAM is enabled and OP-TEE loads a TA, Linux will turn off the clock for the CAAM, leading to a stalled bus transaction [336]. However, the hash of the root of trust cannot be obtained without the CAAM, so we used a workaround. We first boot OP-TEE with the CAAM enabled, fetch the MKVB, and then disable it for further usage in OP-TEE (Linux can still use it). Consequently, software running in the secure world like WATZ cannot profit from the CAAM's accelerated cryptographic operations.

Cryptographic primitives We used the library LibTomCrypt [337] for cryptographic operations in the remote attestation protocol, since OP-TEE already uses it. We decided to use elliptic-curve cryptography (ECC) to reduce the key size for faster transmission and lower processor consumption, while offering the same level of security compared to RSA with a large modulus [338]. We selected the curve *secp256r1* as recommended by the NIST [339], as well as the following algorithms: (i) elliptic curve digital signature algorithm (ECDSA) (256-bit) for the attestation key pair, (ii) elliptic curve Diffie-Hellman ephemeral (ECDHE) (256-bit) for the session keys of the remote attestation protocol, (iii) AES-GCM (128-bit) for data encryption and authentication, (iv) SHA (256-bit) for the anchor in the evidence, and (v) AES-CMAC (128-bit) for the authentication of the messages (*i.e.*, the MAC), to derive the KDK and shared keys.

Deterministic attestation key We also extended OP-TEE to generate an attestation key pair at each boot, in a deterministic manner, based on the hardware root of trust. To accomplish this task, we modified LibTomCrypt in OP-TEE to include a pseudorandom number generator (PRNG) named *Fortuna*, as the OP-TEE's PRNG does not support seeds. Furthermore, we changed the wrapper of LibTomCrypt to generate an ECC key pair with a seed fed into Fortuna. The current implementation of the wrapper for key generations relies on a single instance of PRNG that uses hardware randomness. The ECDSA key pair generation follows a two-step process: (i) deriving the MKVB using the `huk_subkey_derive` function, and (ii) using the resulting value to seed Fortuna and generate the ECDSA key pair using that PRNG instance.

7.5.4 Extension to WASI: WASI-RA

We propose WASI-RA, an extension to WASI for remote attestation, implementing the mechanism designed in WATZ. This enables hosted Wasm applications to control the remote attestation flow. In the remainder of this section, we briefly describe these functions hereafter.

Evidence generation The Wasm runtime exposes two functions for evidence generation to Wasm-compiled applications: `wasi_ra_collect_quote` and `wasi_ra_dispose_quote`. The former function issues evidence based on an anchor provided as a parameter, ensuring the freshness and uniqueness of the generated evidence ($\mathbf{R}_4$). The evidence is returned to the calling application in the form of an opaque handle, abstracting the underlying details of the evidence structure. The latter function requires a handle to previously generated evidence and

is responsible for disposing of the evidence securely. These two functions are deliberately not coupled to the attestation protocol to be used with other transport layers, such as TLS.

Remote attestation protocol The remaining WASI-RA functions implement the remote attestation protocol. The handshake phase, involving the exchange of msg_0 and msg_1, is handled by `wasi_ra_net_handshake` which takes the host address and the verifier's identity (public ECDSA key) as input. The remote party's identity is usually hardcoded in the application, so the code measurement enables the server to detect whether it has been maliciously changed. The function returns an opaque remote attestation context and an anchor, the latter being used for evidence generation. Evidence (msg_2) is sent using `wasi_ra_net_send_quote`, while the secret blob (msg_3) is received through `wasi_ra_net_receive_data` as a variable-sized byte array. Finally, `wasi_ra_net_dispose` is used to dispose of the remote attestation context.

7.6 Evaluation

Our evaluation answers the following questions: (i) are time measurements sufficiently accurate inside TrustZone, and if not, how can we improve this? (ii) what is the performance overhead for compute-bound Wasm-compiled applications in TrustZone? (iii) how do real-world applications, compiled as Wasm, perform in TrustZone? (iv) what are the cost factors of our attestation protocol? (v) what is the overhead of running a machine learning application using the attesation protocol for supplying confidential data set? We respectively answer these questions by assessing the usage of monotonic timers (§7.6.1), using a general-purpose computing-bound evaluation with PolyBench/C (§7.6.3), evaluating SQLite as a real-world embeddable database (§7.6.4), breaking down the cost of our remote attestation operations (§7.6.5) and assessing an end-to-end machine learning scenario using Genann (§7.6.6).

This chapter solely focuses on evaluating WATZ using Arm TrustZone. However, it is worth noting that a comparative analysis of the performance of TWINE and WATZ as two complementary solutions for bridging the cloud-edge continuum has been conducted in a separate paper [5].

Experimental setup All experiments run on an off-the-shelf NXP MCIMX8M evaluation board, equipped with i.MX 8MQ, an Arm Cortex-A53 (1.5 GHz) SoC with the Armv8-A architecture. In the $100 price range, this board supports the hardware root of trust, secure boot, and the CPU fully supports TrustZone. The normal world OS is compiled using BuildRoot 2021.02, with the Linux kernel 5.13 forked by Linaro to ensure compatibility with OP-TEE. The secure world OS runs OP-TEE 3.13. The bootloaders are U-Boot 2020.10-rc2 and Arm Trusted Firmware 2.3.

We show the median and standard deviation of multiple runs for each experiment, as specified with each benchmark. For most experiments, the standard deviation is very small. The native benchmarks are compiled, using GCC 9.2.1 and `-O3` optimisation. The Wasm benchmarks are compiled by WASI-SDK [340], which uses Clang 11 with the same optimisation flag. Our implementation is open-source, and instructions for reproducibility are available on GitHub [17].

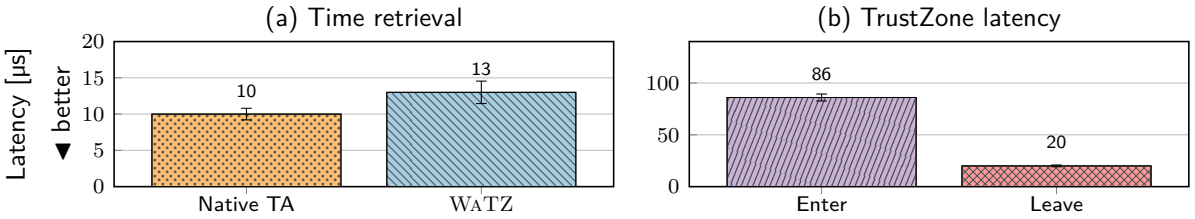


Figure 7.3: The time retrieval and world transition latencies.

7.6.1 Time measurements in TrustZone

The time resolution of the measurements inside TrustZone, offered by default from OP-TEE, is in milliseconds. To achieve nanosecond resolution, we implemented and evaluated the following changes across all involved software stacks to retrieve the same time as provided by the monotonic clock of the Linux kernel in the normal world. For native applications compiled for the normal world, the time is returned using the standard POSIX function `clock_gettime`. However, native applications in the secure world do not have a standardised way to retrieve the same time. Hence, we modified the OP-TEE driver to add a function for passing the value of the monotonic clock to the secure world. We also extended the GP's type `TEE_Time` to measure our experiments with a nanosecond precision, and wired the WASI function `clock_time_get` so Wasm-compiled applications can interact with the high-precision monotonic clock of Linux.

Trusted applications require to specify the heap and stack sizes at compile time. As such, we allocated a heap size of 2 MiB and a stack size of 3 KiB for this benchmark. This amount comprises the memory of the runtime, the bytecode of the Wasm application and the space of the virtual heap and stack allocated by WATZ to execute the Wasm program.

Figure 7.3a shows the latencies to fetch the time in two settings, respectively, from native trusted applications and Wasm in TEE. We ran each experiment 1000 times. The time required in normal world for native and Wasm programs is under 1 µs (not shown). The average latency of retrieving the time in the secure world is 10 µs for native applications and 13 µs for Wasm applications. The increase is due to a transition to the normal world for each query. The benchmarks presented in the following of this section take such latency into account.

Figure 7.3b shows the time to switch between worlds, a frequent operation when an application is partitioned to execute sensitive operations in TrustZone. In WATZ, the server of the verifier invokes functions inside the TEE once received by the TCP server. Our micro-benchmark registers the time in the normal world, before and after the TEE invocation. Similarly, we measured the time in the secure world, upon a function call. We observed an average time of 86 µs to call a function in the secure world, and 20 µs to return, as observed earlier [341].

7.6.2 Startup overhead

Next, we evaluated the startup overhead of Wasm applications loaded into WATZ. For that purpose, we created nine Wasm programs with a code size that varies from 1 MiB to 9 MiB. The AOT-compiled Wasm binaries have been generated by unrolling thousands of loop iterations

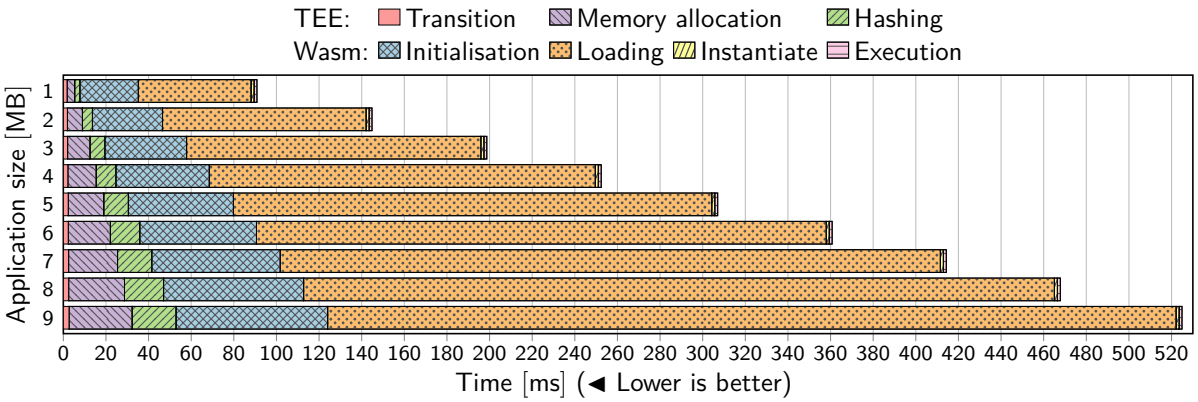


Figure 7.4: The startup breakdown of Wasm applications in WATZ.

to output an amount of code with a size of 1 MiB. That output is replicated multiple times to create the nine variants. Afterwards, we measured the time from the instruction that requests the launch of Wasm application in the normal world until the first instruction in the Wasm application is executed. Finally, the Wasm program stops after the first Wasm instruction since we measure the startup time, preventing the loops from being executed.

We allocated a heap size of 23 MiB, which is the smallest amount of memory to launch the application of the largest size (*i.e.*, 9 MiB). We identified that the overhead of memory, roughly twice the application code size, was bound to the relocation symbols LLVM generated in the AOT-compiled Wasm programs. The underlying Wasm runtime (WAMR) allocates a dedicated structure for each relocation entry. Hence, we determined that the loading operation of an AOT-compiled Wasm binary in the runtime may double the size to allocate, depending on the structure of the code. One way to reduce that overhead is the use of the experimental feature of the AOT compiler called *execution in place*, which generates as few as possible AOT relocations. We leave this optimisation as future work since this feature still has known issues.

Figure 7.4 shows a breakdown summary of the various internal operations when loading the previously mentioned Wasm applications within WATZ. The large majority is dedicated to allocating the memory for the Wasm AOT bytecode (5 %), initialising the Wasm runtime (16 %) and loading the bytecode (73 %). The memory allocation creates two buffers: (i) to store the Wasm AOT bytecode in the secure memory and, (ii) for the heap of the Wasm application. The initialisation of the runtime consists of creating the Wasm runtime environment, initialising the memory allocator, and registering native symbols (*i.e.*, the binding of the native functions imported by the Wasm application). The loading phase parses the bytecode and creates the internal structures required to run Wasm applications, similar to loading a normal world process. This phase notably includes the loading of the relocation entries. Hashing the bytecode takes 4 % of the time. The hash is later embedded in the evidence issued during the remote attestation process. Each of the remaining categories (*i.e.*, the time to transition to the secure world, Wasm instantiation and execution) takes less than 1 % of the startup time. Compared to the baseline Wasm runtime in the normal world (WAMR), the overhead added by WATZ is the transition time and the hashing operation, which represents roughly an increase of 5 %.

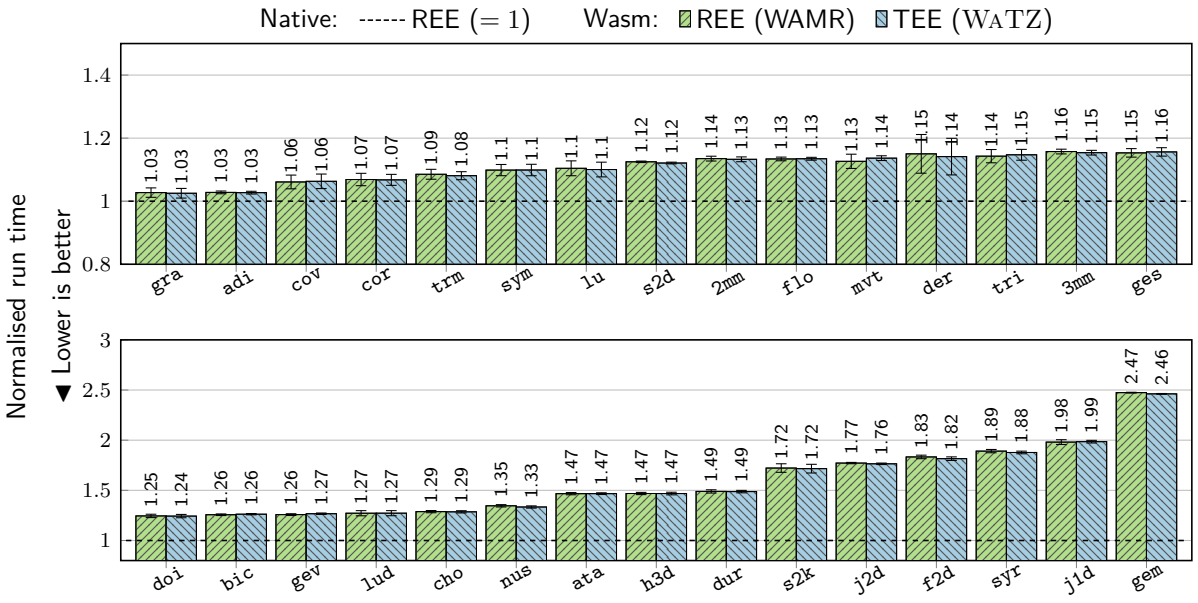


Figure 7.5: The performance of PolyBench/C, relative to native run time (normal world).

7.6.3 Micro-benchmarks: PolyBench/C

PolyBench/C [301] is a CPU-bound benchmark suite commonly used to evaluate compiler optimisations [342] and often used for comparative analysis of Wasm environments [116, 120, 343]. We use all the tests in the PolyBench/C suite (v4.2.1b), executed individually 50 times. We compare the AOT-compiled Wasm benchmarks executed in the normal world (using WAMR) and in the secure world (using WATZ), against native execution (Arm, baseline). Due to memory limitations in the secure world imposed by OP-TEE, we rely on the built-in medium dataset of PolyBench/C for all 30 applications. We allocated a heap size of 12 MiB for WATZ, which is sufficient to execute all the tests of the benchmark suite.

Figure 7.5 shows these results. In both normal and secure worlds, we observe that Wasm's slowdown is on average 1.34× when compared to native execution. Therefore, WATZ does not add additional penalties when executed in the secure world. The difference observed between WAMR and WATZ are insignificant (less than 0.02 %), since TrustZone does not introduce any security mechanism that slows down the computation speed. These results confirm previous work comparing the slowdown of Wasm applications against their native counterpart using JIT and AOT compilation [116]. A qualitative comparison of our results with those found for PolyBench/C in Intel SGX (illustrated in Section 6.6.2) shows that TrustZone does not affect the execution runtime negatively. Unlike our proposal, Intel SGX introduces noticeable performance overheads on AOT-compiled Wasm execution. We think the difference is natural because of the additional security guarantees provided by SGX, with transparent encryption and verification of memory pages stored in volatile memory [163].

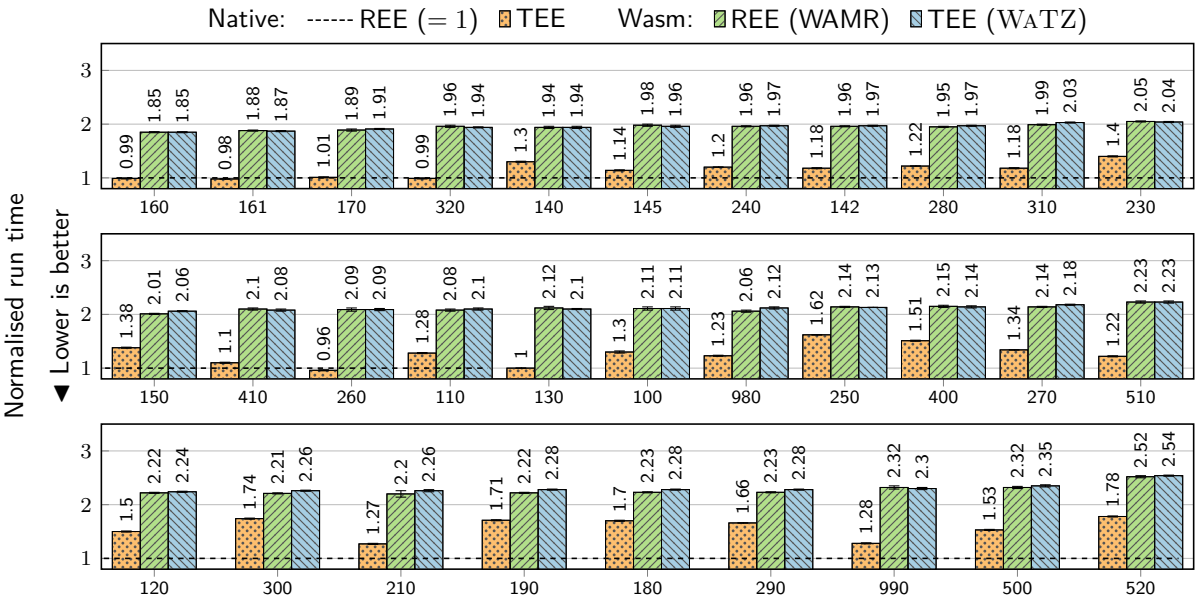


Figure 7.6: The performance of SQLite's Speedtest1, relative to native run time (normal world).

7.6.4 Macro-benchmarks: SQLite

SQLite [302] is a widespread, portable, low memory footprint, full-fledged embeddable database, well suited for constrained environments. Our tests use SQLite v3.36, leveraging its built-in benchmarking suite Speedtest1 [303], similar to the benchmarks suite used with TWINE in Section 6.6.4. We use the native execution of SQLite (ARM64) in the normal and secure worlds as an empirical baseline. As such, we adapted SQLite to be embedded within a TA and run in OP-TEE. We instantiate exclusively in-memory databases, as we have left the implementation of WASI's file system API for future work. Each benchmark in the suite assesses a single aspect of the database engine (*e.g.*, selections with joins, data updates or schema changes). We configured SQLite to use a 2048-page cache of 4 KiB each (for a cache size of 8 MiB), with the default (normal) synchronous mode and the default (delete) journal mode. In addition, we enabled an alternate memory allocator to pre-allocate the memory used by the database. To fit in the restricted memory of the secure world imposed by OP-TEE, we scale down the input dataset to 60 % (argument `--size`). As such, we compiled the TA of WATZ to use 25 MiB of heap memory.

Notice that Arm TrustZone does not impose such memory limitations. Hence the entire dataset can be used as soon as OP-TEE lifts memory restrictions. Provided the memory can be enlarged, we do not foresee any impediment for WATZ to operate on software with large memory footprints, such as deep learning systems for instance. The experiments ran 50 times, and we report medians and standard deviations in Figure 7.6.

Results are normalised against the execution time of Speedtest1 in the normal world, using native execution as a baseline. Overall, our observations are twofold: (i) in the normal world, the slowdown of WAMR is 2.1×, and (ii) in the secure world, the slowdown of native execution and WATZ are 1.31× and 2.12× respectively. As a result, the execution speed overhead of

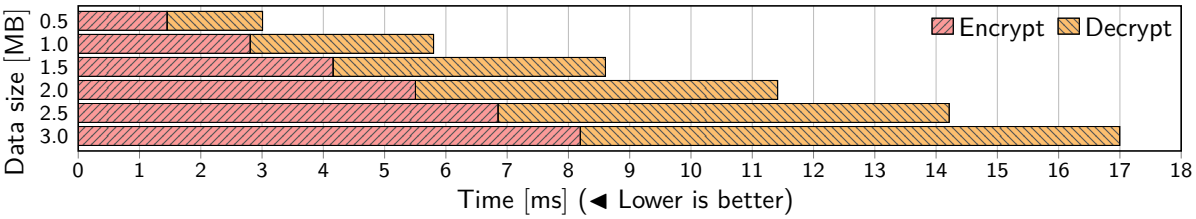


Figure 7.7: The execution time of WATZ's msg_3.

Wasm over native execution in the TEE is 1.61×. This macro-benchmark also demonstrates that WATZ has low to no-overhead execution in TrustZone, compared to Wasm in the normal world. Conversely, the native TA suffers from an overhead compared to the native application in the normal world. These slowdown differences are explained because the compiled binary in the normal world is optimised for using the underlying hardware, unlike Wasm applications that are ahead-of-time compiled from the intermediate Wasm bytecode.

Experiments in Figure 7.6 are ordered by the performance overhead incurred when executed in WATZ. The experiments that have the lowest impact on the performance mostly perform read queries on the database, with an average of 2.04× (*i.e.*, experiments 130-145, 160-170, 260, 310, 320, 410, 510, 520). Instead, most of the experiments located in the bottom plot of Figure 7.6 are write-intensive, by inserting, or updating data in the database, with a slowdown on average of 2.23× (*i.e.*, experiments 100-120, 180, 190, 210, 290, 300, 400, 500, 990). Similarly to the micro-benchmarks, we identify minor discrepancies where the TEE applications are faster (equal to or less than 0.04 %). We also consider them insignificant and are deemed equal to the performance measured in their unsecured counterpart.

In summary, our analysis of Wasm runtime performance in WATZ shows overheads of 1.34× and 2.12× for the micro- and the macro-benchmarks, respectively. We estimate that the security benefits of TrustZone largely compensate for these performance penalties. Besides, we did not observe any noticeable overhead using WATZ compared to WAMR.

Thanks to Wasm, WATZ provides great freedom for the developers who build secure software using various technologies without being bound to the programming restrictions of OP-TEE's SDK. On this topic, we noted that the adaptation of SQLite to be executed in OP-TEE as a native TA has been substantially more laborious than compiling it in Wasm with WASI support. Indeed, the ad-hoc GP API required to modify the source code of SQLite at various locations to replace missing POSIX functions and undefined data structures. On the other hand, WASI-SDK, the framework used to compile C programs in Wasm, provides all the function and structure definitions needed for a seamless compilation. Finally, we leveraged the robust sandbox of WATZ to ensure isolation between each hosted application in the secure world with acceptable costs.

7.6.5 Micro-benchmarks: remote attestation

We now evaluate the remote attestation protocol. For the sake of these tests, the attester (client) and the verifier (server) applications run on the same development board.

(a) Attester	⋆msg_0	◇msg_1	⋆msg_2
Memory management	7 µs	50 µs	5 µs
Key generation	① 236 ms	⑤ 235 ms	—
Symmetric cryptography	—	88 µs	79 µs
Asymmetric cryptography	—	④ 159 ms	⑥ 238 ms
(b) Verifier	◇msg_0	⋆msg_1	◇msg_2
Memory management	52 µs	7 µs	7 µs
Key generation	② 471 ms	—	—
Symmetric cryptography	—	85 µs	80 µs
Asymmetric cryptography	—	③ 236 ms	⑦ 159 ms

⋆Generation of the message. ◇Handling the message.

Table 7.3: The execution time of WATZ's msg_0, msg_1 and msg_2.

We measure the execution time of each message in the remote attestation protocol and highlight the various cost factors. The sizes of msg_0, msg_1 and msg_2 are fixed, and the execution time of their generation and handling are not bound to the size of the Wasm application being measured. On the other hand, msg_3 requires a time that is proportional to the size of the secret blob transferred to the application. As a consequence, we analyse the execution time of the three first messages regardless of Wasm AOT bytecode size in Table 7.3. Figure 7.7 depicts the execution time of the fourth message, according to the size of data to transmit securely. We evaluate the transfer of confidential information, from 512 KiB to 3 MiB. We compiled the attester and the verifier of WATZ to approximately split the heap memory equally, as they are located on the same hardware: 14 MiB and 13 MiB, respectively. We noticed that our implementation requires twice the size of the transmitted data in memory, in each TA. Indeed, we needed to allocate a buffer for the plaintext and a buffer for the ciphertext. For the largest data to transfer (*i.e.*, 3 MiB), this represents 6 MiB in each TA, leading to a memory occupancy of 12 MiB in total. We leave the optimisation of encryption and decryption using a single buffer for future work. In addition to the details below, we report that most of the time is dedicated to complete asymmetric cryptography operations, *i.e.*, keys and signatures generation.

Message 0 The attester generates an ECDHE key pair and sends the public key (① in Table 7.3). When handling message 0, the verifier generates an ECDHE key pair and derives the shared secrets to establish a secure communication channel (②).

Message 1 The verifier signs the ECDHE public keys (③) and authenticates the message with a MAC. The asymmetric signature takes most of the time, up to 2774× the execution time of the MAC, as expected [344]. The derivation of the shared secrets is also performed on the attester's side when handling msg_1 (⑤). Hence, the time the attester takes to generate its keys (①, ⑤) is similar to the time the verifier takes to do the same (②).

Message 2 The attester issues the evidence, which requires a digital signature (⑥), and an authentication with a MAC. Upon reception, the verifier checks the MAC and the evidence signature (⑦). The time to sign (⑥) and verify the signature (⑦) of msg_2 is similar to the time required for the same operations on msg_1 (③ and ④). The same cryptographic operations are performed in both cases, using different data.

Message 3 The verifier generates msg_3 with the secret blob, encrypted and authenticated using AES-GCM. We omitted the time consumed by memory management (less than 1 %). As seen in Figure 7.7, the execution time evolves proportionally between the verifier's encryption and the attester's decryption. Since AES-GCM is symmetric, it is faster than signing the three first messages, starting from 3 ms for 0.5 MiB of encrypted data and reaching 17 ms for 3 MiB.

7.6.6 Macro-benchmarks: remote attestation with Genann

We conclude our evaluation of WATZ with Genann [345], a neural network library extensively used in literature [346, 347]. This library supports feedforward artificial neural networks (ANN) and has zero external dependencies, making it a convenient target to be compiled in Wasm and tested in a constrained memory environment. The benchmark is based on a built-in Genann example, where an ANN is trained on a subset of the Iris dataset [348, 349]. The ANN comprises 4 inputs, 1 hidden layer of 4 neurons and 3 outputs (1 per class). The input dataset includes 50 records per class (file size of 4.45 KiB). We replicated the dataset to reach the breakpoint sizes, from 100 KiB to 1 MiB. Tests are executed 20 times. The attester is launched with Genann as a trusted Wasm application. We allocated 17 MiB for the attester, enough space to handle the Wasm runtime, the attestation with the transfer of the dataset and the heap required by Genann. The remaining memory is allocated for the verifier (*i.e.*, 10 MiB). Once executed, it triggers a remote attestation request to retrieve the dataset, used to train and predict the classification. We use this end-to-end example to demonstrate a real-world workflow using WATZ and assess a few cost factors, such as the impact on the execution time when the size of the confidential information varies. Similarly to the micro-benchmarks of remote attestation detailed in Section 7.6.5, the attester and verifier are co-located on the same device.

Table 7.4 shows the execution time of WASI-RA, the API exposed to the Wasm-compiled applications to request remote attestation and evidence generation. The function retrieving the secret blob is indicated according to the lower and upper bounds of the dataset size. This end-to-end benchmark includes client and server time to generate and handle the messages, the overhead caused by the socket connection and the penalty of the normal and secure world switching.

Most of the execution time is spent on the handshake: msg_0 and msg_1 handle the key generation and half on the asymmetric operations, as seen in Table 7.3. The generation of the evidence is the second most time-consuming operation due to the digital signature (⑦ in Table 7.3). The sending of the evidence consumes only a marginal time.

Lastly, we evaluated several dataset sizes to assess the execution time of the function that receives the confidential information. We report an execution time ranging from 168 ms for 100 KiB up to 209 ms for 1 MiB. According to the micro-benchmark in Section 7.6.5, the cryptographic operations of msg_3 takes a negligible amount of time: 5.8 ms for 1 MiB of data, unlike

(a) Fixed-size operations				(b) Variable-size operations		
handshake	collect_quote	send_quote	Σ[(a)]	Size	receive_data	Σ[(a)]+Σ[(b)]
1.34 s	239 ms	1 ms	= 1.58 s	0.1 MiB	168 ms	= 1.75 s
1.34 s	239 ms	1 ms	= 1.58 s	0.2 MiB	172 ms	= 1.76 s
1.34 s	239 ms	1 ms	= 1.58 s	0.3 MiB	175 ms	= 1.76 s
1.34 s	239 ms	1 ms	= 1.58 s	0.4 MiB	182 ms	= 1.77 s
1.34 s	239 ms	1 ms	= 1.58 s	0.5 MiB	185 ms	= 1.77 s
1.34 s	239 ms	1 ms	= 1.58 s	0.6 MiB	192 ms	= 1.78 s
1.34 s	239 ms	1 ms	= 1.58 s	0.7 MiB	196 ms	= 1.78 s
1.34 s	239 ms	1 ms	= 1.58 s	0.8 MiB	200 ms	= 1.78 s
1.34 s	239 ms	1 ms	= 1.58 s	0.9 MiB	205 ms	= 1.79 s
1.34 s	239 ms	1 ms	= 1.58 s	1.0 MiB	209 ms	= 1.79 s

Table 7.4: The execution time breakdown of the WASI-RA API, segmented into (a) the fixed-size operations inherent to the remote attestation process and (b) the variable-size operations dependent on the size of the secret blob.

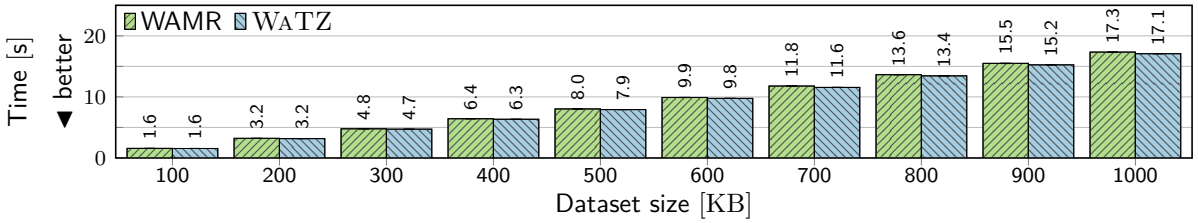


Figure 7.8: The execution time of the training phase.

this macro-benchmark. This difference is due to the attester waiting for the end of the evidence verification on the verifier's side, reported for lasting 159 ms in Table 7.3 (⑦). We confirmed this was the cause by waiting before receiving the data, which leads to a reception duration of only 70 ms for 1 MiB. This points out the importance of having appropriate hardware for cryptographic operations on the verifier's side.

Further, we evaluate the training time of the model for different dataset sizes in Figure 7.8. For the baseline experiment in the normal world, the dataset is fetched from a regular file. In WATZ, it is read via a secure channel established during remote attestation. WATZ achieves a 1.4 % speedup, roughly matching previous experiments with no runtime overheads. The time difference between the two systems is yet to be further investigated. Indeed, using the same AOT binary and supplying the same data to Genann's training function yields better performance in OP-TEE. We excluded the cause of better threads scheduling, since OP-TEE relies on the Linux scheduler, located in the normal world. Finally, we report how the performance obtained here confirms similar results in literature [127], where an equally powerful TrustZone hardware (HiKey Arm Cortex-A53, at 1.2 GHz) is used to benchmark remote attestation protocols.

7.7 Security analysis

This section provides a security analysis of the software components and security measures employed within WATZ and discusses their strengths and limitations.

Secure boot The first-stage bootloader (ROM) and the one-time programmable fuses (eFuses) ensure that the board is booting the secure OS in a genuine state. Assuming an attacker replaces the image of the trusted OS, the software signature will no longer match the binaries, which aborts the boot sequence. Since the signature's public key is stored in the eFuses, it cannot be altered to install a compromised system. Consequently, only a genuine version of the secure kernel OS can access the root of trust. Nonetheless, the system does not protect against rollback attacks. This can be locally mitigated using monotonic counters bound to the hardware, and remotely mitigated by checking the version of WATZ in the evidence during remote attestation requests. While not available in the SoC of our evaluation board, the security guarantees of WATZ may be further extended using a measured boot in addition to the secure boot. Measured boot is a security mechanism that enables the attester to collect the code measurement of every loaded software (*e.g.*, bootloader, trusted OS) and accumulate them in special registers. Once these values (considered additional claims) are collected, they are signed and exported, typically using a Trusted Platform Module (TPM). These claims may be later incorporated into evidence produced by WATZ as a system-wide measurement. Consequently, verifiers may also appraise the startup components to identify a potentially hijacked secure boot.

Trusted Wasm runtime The runtime offers execution and attestation facilities similar to Intel SGX. The normal world requests WATZ to start Wasm applications and, as a result, they are executed in isolation inside Arm TrustZone. Due to the memory sandboxing inherited from Wasm, malicious applications loaded in WATZ cannot compromise the secure world or access resources owned by other TAs. The threat model is different from the regular TAs in OP-TEE, because a signing key is needed to deploy software in the secure world, which prevents the execution of arbitrary software [350]. Moreover, the communication of the signing key to allow third-party developers to execute trustworthy code in OP-TEE may lead to impersonation attacks on already deployed applications, ultimately leaking secrets stored in persistent storage [351]. As such, the Wasm sandbox offers an additional safe and efficient isolation mechanism to host software made by different distrusted parties. The reliance on a runtime has the downside of increasing the size of the TCB. Furthermore, zero-day vulnerabilities found in the Wasm sandbox may lead to data leakage or arbitrary code execution, because the isolation offered by Wasm is implemented in software, as opposed to the isolation mechanisms of SGX enclaves. We note this risk also applies to the TAs of OP-TEE, since vulnerabilities found in the trusted OS may lead to privilege escalations and, therefore, weaken the isolation of the TAs. We mitigate this issue by providing the version of WATZ in the evidence, so the verifier can determine whether the deployed runtime is up to date, *i.e.*, detect whether the attester has mitigation patches applied, similar to how previously discovered security vulnerabilities are mitigated in Intel SGX.

Remote attestation protocol We evaluated the correctness of our remote attestation protocol with Scyther [352], a state-of-the-art formal analyser. This tool is known to analyse security

protocols, security requirements and identify vulnerabilities formally. Scyther is based on the protocol semantics model for the Dolev-Yao intruder model [353], which assumes that an adversary has complete control over the communication channel, but cannot subvert cryptographic techniques. We configured Scyther to check the secrecy of the private session keys, the shared secret and the secret blob. Besides, we verified these claims of authentication: *aliveness*, *weak agreement*, *non-injective agreement*, *non-injective synchronisation* and *reachability* (*i.e.*, the protocol ended on both parties). While these terms are omitted for conciseness [354, 355], they represent essential characteristics of a security protocol. Scyther revealed no attack or flaw in our proposal. The verification script is available in the repository of WaTZ [17].

7.8 Synthesis and next steps

WaTZ is the first Wasm runtime executing entirely inside Arm TrustZone with full support for remote attestation, optimised explicitly for Wasm to establish trust on hosted applications. In many ways, WaTZ offers a model similar to Intel SGX while overcoming the limitations of Arm TrustZone, extending the paradigm of a single trusted world into fully isolated and mutually distrusting trusted applications. It comprises a trusted environment for running unmodified Wasm applications that cannot be tampered with. It relies on a hardware root of trust and a secure boot, shielded by Arm TrustZone. Our WASI extensions for remote attestation and the attestation protocol establish trust on remotely executed Wasm applications inside WaTZ and securely provision confidential data, such as shared keys to join a channel or decrypt a configuration file. Our extensive experimental evaluation assesses the costs of the WaTZ mechanisms with typical tasks IoT devices carry out. As a result, our implementation is lightweight, achieving results on par with Wasm running in the normal world.

We have a strong commitment to open-source and reproducibility: some of our improvements are already pushed to the open software projects we depend on, and we intend to keep doing so. The code of WaTZ and the experiments presented in this paper are available on GitHub [17].

Building upon the proposed attestation protocol, WaTZ can be further enhanced by implementing the missing GlobalPlatform API to establish TLS communication channels with other devices. This extension would enable WaTZ to integrate with existing standardised communication systems and security protocols. In the forthcoming part of this manuscript, we aim to bridge the gap between our trusted and attestable Wasm runtimes executing in TEEs and widely-adopted communication frameworks, such as the MQTT message queuing service in conjunction with the TLS protocol. To achieve this objective, the next chapter explores the security requirements and properties necessary for tightly coupling attestation and security protocols to convey trust through these existing protocols and providing a comprehensive implementation for Twine.

Part III

Establishing trust using WebAssembly for network communication

Chapter 8

Trustworthy distributed systems with WebAssembly and TEEs

The previous part explored the design and implementation of trusted runtimes executing applications compiled into WebAssembly (Wasm) for isolating and securing computations in two well-established trusted execution environments (TEEs). Although these trusted environments introduced attestation primitives, conveying trust evidence to third-party devices and services remains a challenging task when integrating into traditional software stacks. Building upon these trusted runtimes, this chapter demonstrates a practical application of TEE, where a standard communication service (Moquitto, an MQTT broker) and a security protocol (TLS) are adapted for trustworthy communication among many distributed and mutually distrusted nodes.

Chapter outline

8.1 Introduction . . . 138
8.2 Publish/subscribe systems . . . 139
8.2.1 Publish/subscribe with TEEs . . . 140
8.3 Communication channel and attestation binding . . . 141
8.3.1 Binding communication protocols and attestation using TEEs . . . 142
8.4 Design considerations . . . 143
8.4.1 Threat model . . . 143
8.4.2 Security requirements . . . 144
8.4.3 Trusted primitives . . . 144
8.5 Architecture . . . 145
8.5.1 Attesting communication channels . . . 146
8.5.2 Securing publish/subscribe systems . . . 147
8.5.3 Implementation . . . 148
8.6 Evaluation . . . 150
8.6.1 Establishing new connections . . . 150
8.6.2 Messages throughput . . . 151
8.6.3 Scaling the publishers . . . 152
8.7 Synthesis . . . 153

8.1 Introduction

Publish/subscribe (pub/sub) systems [356] have become foundational for seamless intercommunication of a wide range of devices, from IoT ecosystems to large-scale cloud-based services, *i.e.*, the cloud-edge continuum [357]. These systems enable efficient and scalable data distribution among distributed entities by decoupling data producers from data consumers. Given their widespread adoption, notably in cloud computing [358–360], several pub/sub systems have been proposed with the clear goal of providing additional privacy guarantees [361]. However, the nodes participating in these systems are implicitly trusted, relegating security concerns primarily to the protection of communication channels or leveraging heavyweight cryptographic primitives [362]. This limited security approach leaves data on the processing components vulnerable to potential threats, especially in decentralised and heterogeneous environments. Notably, high-privileged actors within the pub/sub nodes, *i.e.*, operating system or hypervisor, may compromise the confidentiality and integrity of data. Similarly, they could leak critical cryptographic material, *e.g.*, private keys of certificates. Leaking certificate keys is especially concerning as these keys serve as the foundation for the authentication process among pub/sub participants, thereby putting user privacy and data integrity at stake.

In both the consumer market and cloud providers, trusted execution environment (TEE) present a solution to strengthen the integrity and confidentiality of data in use, especially on nodes that may not be inherently trustworthy. TEEs provide *enclaves*, *e.g.*, hardware-protected memory regions, where sensitive computations are completely isolated from other software executed on the same platform. Such secure enclaves significantly elevate the security posture of systems like pub/sub, safeguarding not just the communication but also the processing and data on such nodes from malicious actors. As a keystone feature of TEEs, attestation enables a remote entity to verify the authenticity, configuration, and state of a trusted environment using cryptographic proofs, ensuring the enclave runs the intended software without being tampered with or compromised. In pub/sub systems deployed over untrusted infrastructures, attestation ensures data and its processing within the TEE enclave are kept confidential and untampered.

Nonetheless, developing trusted applications (TAs) for TEEs is challenging due to their specific programming paradigms and SDKs, requiring massive efforts when writing or porting existing software [363]. In addition, while many pub/sub systems exploit TEEs to protect data in use (covered in Section 8.2.1), they often tie closely to specific TEE architectures, limiting their applicability in heterogeneous environments (*e.g.*, different CPU architectures) considered in this thesis. Beyond this limitation, current communication protocols fail to transport attestation proofs, do not maintain full privacy of attestation, or cannot expose X.509 certificates issued by global authorities (discussed in Section 8.3.1). This leads to the use of ad-hoc implementations that fall short in offering a consistent solution across the cloud-edge continuum.

This chapter addresses the challenges associated to writing secure and portable pub/sub systems using Wasm, a portable compilation target and binary instruction format. Wasm's hardware-agnostic design abstracts the complexity of heterogeneous TEE environments, making it an particularly suitable compilation target for pub/sub systems across the cloud-edge continuum. We further protect the communication channels by extending the industry standard TLS protocol for embedding attestation evidence in the TLS handshake while maintaining compatibility

with the original specifications. This ensures the authenticity of the executing code of parties within the pub/sub system, mitigating the risk of malicious entities impersonating or modifying these components. Additionally, certificate keys are safeguarded within the TEEs, preventing potential leaks. We developed a proof-of-concept that encapsulates the implementation of a standard pub/sub broker and a TLS library, enabling the termination of TLS channels directly in the TEE, which has been proven complex in prior work [364]. We achieve this by using Wasm as a compilation target for both software, limiting the number of code changes required to make them compile and run within TEEs. While our prototype is focused on cloud environments by leveraging Intel SGX, we outline the trusted primitives required for our proposal to be compatible with other platforms, including edge and IoT devices.

We summarise our contributions as follows: (i) a unified strategy that secures a standard pub/-sub system using TEEs with attestation for security, and compiled in Wasm for seamless cloud-edge communication while minimising code changes, (ii) an extension to the TLS communication protocol, facilitating the confidential exchange of attestation evidence, thereby affirming the authenticity of actors within the pub/sub system and (iii) an open-source implementation of a pub/sub broker using Intel SGX for cloud environments, with a suite of benchmarks aimed at evaluating the impacts of Wasm, the TEE and attestation, in comparison with state-of-the-art work. Our evaluation reveals that our system delivers messages at 1.55× slower than baseline throughput, yet provides portability and the robust security guarantees of TEEs.

8.2 Publish/subscribe systems

A publish/subscribe system (often called pub/sub) is an asynchronous architecture for message passing that connects two different types of entities: publishers and subscribers. Publishers send messages (or events), being unaware of who is interested in receiving such messages. Messages sent are usually marked with an arbitrary type or a category. Subscribers express their interest in specific types of messages and receive them when they are published. Pub/sub is commonly used in event-driven distributed systems such as the Internet of things (IoT), since they simplify the communication between different entities.

Pub/sub messages are usually passed from publishers to subscribers using an intermediary broker. Brokers receive messages from publishers and forward them to subscribers who have expressed interest in the corresponding topics. Message brokers are responsible for routing and delivering messages to the appropriate subscribers. Examples of message brokers include Apache Kafka [365], RabbitMQ [366], Eclipse Mosquitto [367], and cloud-based pub/sub services like Amazon Simple Notification Service (SNS) [368] or Google Cloud Pub/Sub [359].

One of the key advantages of pub/sub is decoupling since publishers and subscribers do not need to know each other. Pub/sub systems can also be designed to be highly scalable and fault-tolerant by using replicated, interconnected brokers. This flexible architecture allows components to be added or removed without affecting the system.

Security in pub/sub systems is dealt with when publishers and subscribers connect to their brokers. Brokers then usually implement authentication and access control before publishing or

subscribing to messages. They can also establish encrypted connections (*e.g.*, TLS) to implement message privacy. By implementing attestation, we offer a stronger guarantee for all the components. Publishers, subscribers, and brokers are guaranteed to be who they say they are, and all can ensure that their counterparts execute appropriate (correct) software.

8.2.1 Publish/subscribe with TEEs

In distributed systems, pub/sub mechanisms have consistently gained traction, supporting a variety of applications and architectures. Many approaches have been conducted in research to bring dependability and safety regarding pub/sub systems, and in many directions [369]. An explored aspect is encryption schemes of data transferred through the brokers, preserving the privacy of the communicated information [370–376]. Cryptographic-based privacy protection schemes focus on encrypting events and subscriptions, and then performing ciphertext matching between them. However, they often suffer from scalability issues as matching time complexity grows with the number of subscriptions, leading to diminishing performance.

Recent research has explored Intel SGX's potential for confidential data processing. SCBR [377] showed that TEEs can offer superior performance compared to cryptographic variants through a custom content-based routing engine within an SGX enclave. PubSub-SGX [378] introduced a scalable approach using a load balancer that manages multiple matchers, each operating within individual enclaves. MagikCube [379] added an authentication service to the pub/sub system, enhancing broker trust among publishers and subscribers using SGX. Finally, Pei *et al*. [380] further refined this paradigm by optimising subscription matching times using cryptographic methods, with SGX facilitating comparison tasks but does not disclose how secrets are exchanged. In contrast with prior work, our proposal prioritises establishing an initial trust across pub/sub participants by mutual attestation. We do it by encapsulating conventional pub/sub systems within TEEs, ensuring genuine execution environments and trustworthy implementations.

Other prior studies have chosen a different strategy using Arm TrustZone. In this setup, the device is divided into the *normal world* (the standard OS) and the *trusted world* (the TEE). MQT-TZ [381] moved the broker's data management within this trusted world. Publishers and subscribers negotiate symmetric keys with the broker, which are generated inside the TEE. This approach ensures that data is encrypted during transmission. Conversely, our proposal enhances the threat model of MQT-TZ by relying on attestation of the pub/sub system and hosting the TLS endpoint within the TEE, avoiding handling cryptographic materials outside the TLS protocol.

Comparison Table 8.1 offers a comprehensive summary of state-of-the-art proposals for securing pub/sub systems with TEEs, focusing on features relevant to our study. Each feature can either be not disclosed (❓), missing (○), partially (◑), or fully (●) available. The definitions of the features are given as follows: (i) **Communication protocol**: protocol used by peers to interact with the broker, (ii) **Fully enclaved broker**: broker operates within the TEE instead of only securing specific components within the secure environment, (iii) **Peer authentication**: broker authenticates the peers while establishing communication, (iv) **Peer attestation**: broker attests the peers while establishing communication, (v) **Broker authentication**: peers authenticate the broker while establishing communication, (vi) **Broker attestation**: peers attest the

Features	SCBR	PubSub-SGX	MagikCube	MQT-TZ	Pei *et al.*	This work
Communication protocol	TLS	TLS	TLS	TLS	?	TLS
Fully enclaved broker	●	●	●	○	○	●
Peer authentication	○	●	●	●	?	●
Peer attestation	○	○	○	○	?	●
Broker authentication	●	●	●	●	?	●
Broker attestation	◑	●	●	○	?	●
Persistence of messages	○	●	○	○	○	●
Idiomatic pub/sub architecture	○	●	●	●	●	●
Open source	○	○	○	●	○	●
TEE technology	SGX	SGX	SGX	TrustZone	SGX	*Agnostic*

Table 8.1: The comparison of the state-of-the-art pub/sub systems shielded by TEEs.

broker while establishing communication; ◑ means that only publishers are attested, (vii) **Persistence of messages**: system's feature to store messages for future delivery (*e.g.*, when subscribers might be temporarily offline), (viii) **Idiomatic pub/sub architecture**: system adheres to the principles of the pub/sub paradigm, (ix) **Open source**: implemented solution is freely available to the public via an open-source repository, and (x) **TEE technology**: denotes the TEE used by the proposed system, or its capacity to operate agnostically across various TEEs.

8.3 Communication channel and attestation binding

Over the past decades, several standards have emerged to establish trustworthy communication between two parties. Among them, the protocol Transport Layer Security (TLS) stands out as the most prominent protocol for such tasks. This cryptographic protocol ensures confidentiality, integrity, and authentication for secure communication. A typical TLS session begins with a phase known as the *handshake*. The handshake acts as a negotiation process, where the two parties decide on encryption settings and authenticate one another before exchanging secure data. More precisely, the latest version of the handshake (TLS 1.3 at the time of this writing) is composed of (i) the *ClientHello* sent by the client, notably containing the supporting cipher suites, key agreement and anti-replay mechanisms, (ii) the *ServerHello* replied by the server, detailing analogous parameters, and indicating the handshake's end, and finally (iii) the handshake wraps up as the client reciprocates with a similar acknowledgement, accompanied by data from a higher-level protocol, that is wrapped within TLS, *e.g.*, HTTPS. Additionally, during steps 2 and 3, both entities can opt to present an X.509 certificate. These serve as identity validators, embedding trust through a chain of digital signatures rooted in trusted certificate authorities (CAs). Building on this, many existing attestation protocols that bind with TLS typically augment the handshake, the certificate, or a combination of the two, facilitating the negotiation of security parameters and the exchange of attestation evidence.

Channel binding [382] ensures that an entity participating in a secure communication, typically via protocols such as TLS, is indeed the entity that has undergone attestation. It establishes

that the communications are with the attested environment, preventing possible relay attacks where a malicious party might relay attestation challenges to a genuine system and then claim the legitimate evidence as their own. To secure trusted communication, attestation evidence is typically integrated into the early phases of the communication protocol. However, combining attestation with these protocols introduces challenges, including increased latency and additional trade-offs like binding the system to a particular TEE technology.

We propose a refined handshake with minimal enhancements to address the limitations of prior approaches. Instead of embedding attestation in custom X.509 certificate fields, we leverage the encrypted extension of the handshake. This approach prevents additional roundtrips with brokers using certificates issued by recognised CAs, an aspect overlooked in previous research.

8.3.1 Binding communication protocols and attestation using TEEs

Research has extensively explored the integration of secure communication protocols with attestation. While our focus primarily lies on solutions leveraging TEEs, we also recognise the significant contributions from prior work that leveraged Trusted Platform Modules (TPMs) as trust anchors [140–142, 383–389]. Readers can refer to [131, 132] for a comprehensive review of these works. Our analysis omits explicitly works that discuss attestation through TEEs but do not bind attestation evidence to a communication channel [103, 196, 390, 391].

Intel proposed a remote attestation protocol for key exchange based on SIGMA [149, 333], binding SGX enclaves with communication channels. Subsequent solutions aimed to establish trusted communication channels with enclaves using custom message exchanges [105, 127], including WATZ (in Chapter 7). While these initial efforts in remote attestation provided valuable insights, their ad-hoc nature makes them challenging to integrate into existing software. In contrast, our approach harnesses TLS, the leading industry standard for secure communication.

Recent research [121, 131, 132, 212, 392] proposed using TLS for communication and modifying the X.509 certificates to include attestation-related fields. Although this approach leverages the protocol's standardisation to address the earlier concern, it ties the attestation mechanism directly to the certificates exposed by the endpoints. This direct connection implies that certificates must be dynamically generated, which restricts them from being signed by global CAs like *Let's Encrypt*, commonly used for domain certificates. Opting for a different route, we used the encrypted extensions of the TLS handshake for carrying evidence of the server. Consequently, our approach is compatible with certificates endorsed by global CAs. This is particularly beneficial if the endpoint owns a separate network-level identity, like a domain name.

Comparison Table 8.2 offers a comprehensive summary of cutting-edge research dedicated to binding communication channels with attestation, focusing on features relevant to our study. Each feature can either be not applicable (—), not disclosed (❓), missing (○), partially (◑), or fully (●) available. The definitions of the features are given as follows: (i) **Baseline protocol**: protocol upon which the proposal is built, (ii) **No change in TLS specification**: proposal respects the TLS specifications, ensuring compatibility with pre-existing TLS implementations, (iii) **Attestation privacy**: protocol maintains confidentiality of attestation evidence, (iv) **Mutual attestation**: communicating entities engage in a mutual attestation process, (v) **Evidence**

Features	SGX EPID	Shepherd *et al.*	WaTZ	RA-TLS	Palæmon	TSL	This work
Baseline protocol	Custom	Custom	Custom	TLS	TLS	TLS	TLS
No change in TLS specification	—	—	—	●	●	●	●
Attestation privacy	○	●	○	○	●	●	●
Mutual attestation	○	●	○	●	○	●	●
Evidence per session	●	●	●	○	○	●	●
Endpoint in enclave	●	●	●	●	●	○	●
Attestation privacy	○	◑	○	◑	?	◑	●
TEE-agnostic	○	◑	◑	○	○	◑	●
Support global CAs	—	—	—	○	○	○	●
Open-source	●	○	●	●	○	○	●
TEE technology	SGX	TrustZone	TrustZone	SGX	SGX	*Agnostic*	*Agnostic*

Table 8.2: The comparison of the state-of-the-art channel binding solutions.

per session: each communication session is uniquely associated with specific attestation evidence, (vi) **Attestation Privacy**: all attestation-related messages are encrypted; (◑) means that only a portion of the messages remains confidential, (vii) **TEE-agnostic**: not bound to specific programming language or TEE-specific SDK; (◑) means theoretical approach proposed, but no agnostic implementation, (viii) **Endpoint in enclave**: endpoint application fully resides in the TEE, (ix) **Support global CAs**: broker-displayed certificates can be vouched for by globally CAs, (x) **Open source**: implementation is available to the public via an open-source repository, and (xi) **TEE technology**: denotes the TEE that the attestation API currently supports.

8.4 Design considerations

We explain the threat model of our design, highlight its security requirements and cover the trusted primitives that must be available on a system to support the proposal of this chapter.

8.4.1 Threat model

Our proposal leverages the protection offered by TEEs for securing the execution of applications and enforcing strong isolation against powerful attackers, such as the OS or the hypervisor. Given that our proposal is TEE-agnostic, we highlight the minimal requirements to uphold trust in the pub/sub system and attestation mechanism. We assume the application code can be inspected but cannot be subverted. Data in use remains confidential and cannot be read unless granted by the TEE. The hardware and software strictly required to run a TEE instance are considered trusted. Furthermore, the TEE offers remote attestation to verify the trustworthiness of the device and its software. Although we do not address side-channel or denial-of-service attacks [393–395], there exist measures for these in various TEE designs [396].

The OS follows an honest-but-curious threat model, posing no threat to the trusted environment but seeking to collect sensitive data. Consequently, it can monitor all encrypted communica-

tion with the pub/sub system. If the OS acts maliciously, the trusted computing base (TCB) remains confidential and doesn't malfunction, although it may become unresponsive. The TAs are designed to disregard abnormal responses and halt execution in such cases.

We presume that the Wasm runtime is implemented correctly and does not contain vulnerabilities. The Wasm runtime acts as a shim library by encapsulating Wasm applications and uses the trusted API from the TEE SDK for system interactions or sanitises the interaction with the untrusted OS when no secure option is available. While side-channel attacks might target Wasm and TEEs [292], they fall beyond the scope of this study.

As we embed a TLS library and enhance it to integrate attestation, we suppose the correct implementation of the cryptographic primitives. Besides, we presume that standard cryptographic techniques cannot be subverted and are hardened against side-channel attacks.

8.4.2 Security requirements

We propose a series of security requirements for establishing trusted communication channels between the actors of pub/sub systems.

- $\mathbf{R}_1$. **Support global CAs certificates**: brokers shall support exposing certificates issued by globally recognised CAs.
- $\mathbf{R}_2$. **Trust assurance**: pub/sub actors shall attest the TCB of the participant they connect with and must be similarly verified in return.
- $\mathbf{R}_3$. **Channel and attestation bindings**: communication channels must be linked to newly created attestation evidence, preventing replay or collusion attacks [131].
- $\mathbf{R}_4$. **Pub/sub privacy**: all the data bound to the pub/sub system shall be inaccessible to outside actors, including the OS, kernel and hypervisor of the broker and peers.
- $\mathbf{R}_5$. **Pub/sub narrow scope**: the peers shall only publish or subscribe to the topics as required for their needs.

8.4.3 Trusted primitives

We identified a set of trusted primitives that any system must support to host our proposal securely. These primitives, enumerated below, are provided by the Wasm runtime or the TEE. Platform-independent concerns, including encryption and pub/sub protocol logic, are addressed using dedicated software compiled into Wasm and self-contained within the proposed approach.

- **Isolated execution context**: this ensures the runtime remains secure and unaffected by any other applications running concurrently on the system. All major TEE manufacturers support at least one Wasm runtime, though the isolation paradigm and threat model vary.

- **Attestation mechanism**: the TEE must expose two primitives to the Wasm runtimes: *generate* and *verify*. The former primitive generates evidence for proving the trustworthiness of the secure environment with additional data attached, such as a nonce and a public key, while the latter confirms the validity of this proof.
- **Network communication**: the Wasm runtimes require access to a network API that handles socket operations and the transfer of data.
- **File system access**: if the pub/sub broker needs to store publications for future delivery, the system should offer secure ways to save and access these files.

8.5 Architecture

We designed our proposal as a versatile system capable of running on numerous processor architectures. The system isolates security-sensitive pub/sub operations of peers and brokers, ensuring the authenticity of connecting machines using TEEs and mutual attestation. A key aspect of our design is the adoption of Wasm, facilitating cross-platform support across various TEE architectures. More specifically, we rely on trusted Wasm runtimes, such as Twine and WaTZ, to host a secure pub/sub system with its associated dependencies such as TLS libraries, thus covering the large spectrum of the cloud-edge continuum.

Figure 8.1 illustrates the key components and entities within our pub/sub solution. Serving as the central hub, the broker first acquires a certificate from a global CA for its TLS endpoint (①). Peers then initiate secure communication with the broker via our enhanced TLS handshake,

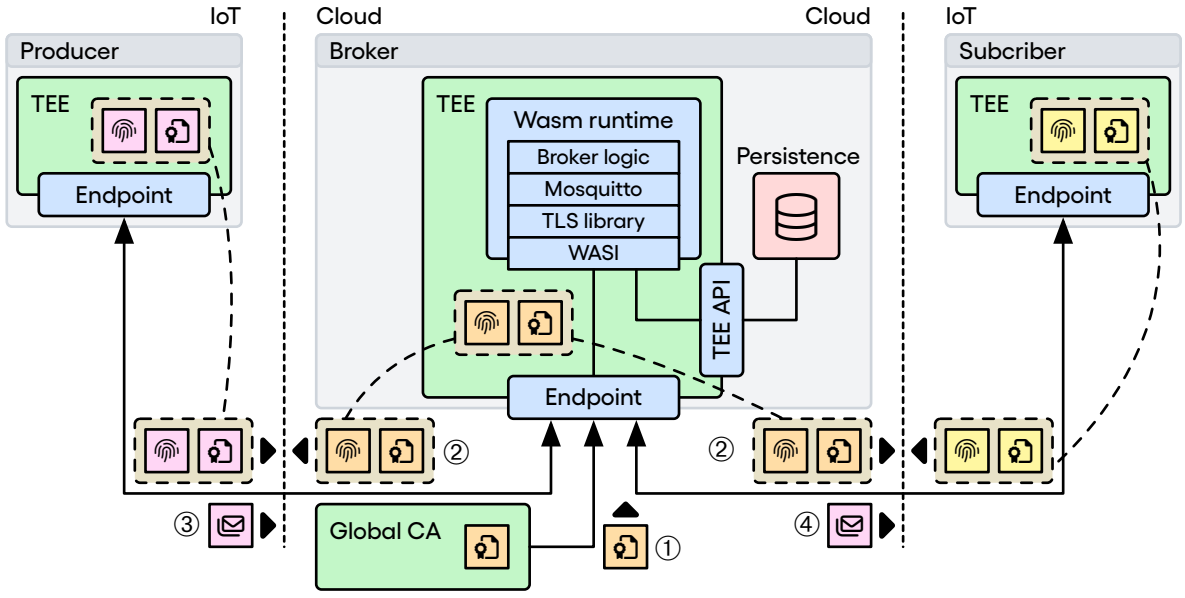


Figure 8.1: The architecture of the proposal. The X.509 certificates (🔒) and evidence (🫆) are represented with colours corresponding to the owning actor. The green boxes (▢) are trusted entities, the red box (▢) is untrusted, the blue boxes (▢) are software components and the brown boxes (▢) are associated data.

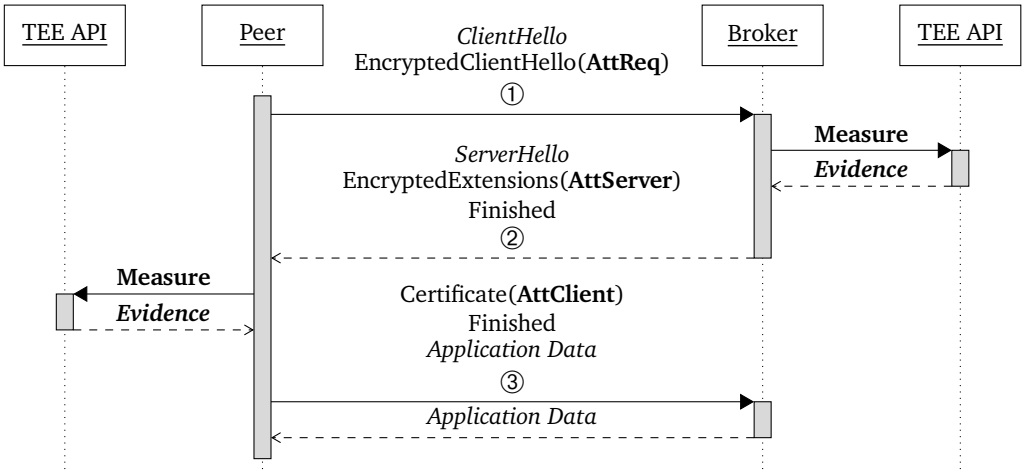


Figure 8.2: The enhanced TLS 1.3 handshake with attestation. New elements are in bold.

performing mutual remote attestation by exchanging their certificate for authentication and attestation evidence (detailed in the next section). This ensures that the broker and peers are trustworthy (②). The publisher generates and transfers data to the broker's TEE (③). The broker, in turn, relays the data to the subscriber's TEE (④). Publishers and subscribers use the same TEE technology stack in their respective TEEs, although omitted in the figure for clarity.

8.5.1 Attesting communication channels

We enhanced the TLS protocol to integrate the exchange of attestation evidence when a peer is communicating with the broker. This exchange of information occurs in the TLS handshake, as depicted in Figure 8.2. Our enhancements are highlighted in bold. The first message of the handshake (①) is sent by the peer, which comprises an encrypted attestation request (**AttReq**), indicating that the peer is establishing a TLS channel requiring attestation. Unlike prior work, our protocol does not specify TEE architectures in this message, as our solution supports the verification of all the types of TEEs that can be used in the broker. Similarly, our protocol is the only proposal that studies the encryption of the attestation request, as further explicated below.

Upon receiving message (①), the broker answers with a message (②) composed of its evidence (**AttServer**), freshly generated and bound to the TLS session. It is worth noting that we deliberately have chosen not to embed attestation data within the broker's X.509 certificate so the endpoint can use certificates endorsed by globally recognised CAs, satisfying $\mathbf{R}_1$.

Subsequently, the peer verifies whether the evidence of the broker is genuine and checks if the evidence is bound to the current TLS session (to counter reuse attacks). Moreover, it compares the code measurement of the broker to a known reference value, indicating that the code of the broker is trusted. Since a global CA issues the broker's X.509 certificate, the peer verifies that the domain name of the broker matches the identifier exposed in the certificate [397]. If these conditions are fulfilled, the peer issues its evidence (**AttClient**) and sends it to the broker

(③), also bound to the TLS session and embedded within the peer's X.509 certificate using an ad-hoc field, as the TLS handshake lacks an extension point for last message.

Lastly, the broker verifies whether the evidence of the peer is genuine, and checks if the evidence is bound to the TLS session. Similarly, the broker ensures that the code measurement of the peer matches a known reference value, indicating that the code of the peer is trusted. Upon successful mutual attestation, confirming the trustworthiness of both TEE instances and fulfilling $\mathbf{R}_2$, the TLS handshake concludes, and the applications may start pub/sub communication.

Integrating the attestation mechanism in the TLS handshake has many benefits: the execution time is optimised since no additional round-trips are necessary, and the implementation respects the TLS 1.3 standard by using extension points as designed by the protocol. Besides, we strongly bind freshly generated evidence to individual TLS sessions, using unique TLS keying materials computable by both ends of the communication channel (RFC 5705) [398], satisfying $\mathbf{R}_3$. This strong binding prevents replay and collusion attacks, which would affect published past evidence otherwise. Since our pub/sub system and TLS library are compiled in Wasm, evidence binding within TLS handshakes is portable across different TEE architectures.

We use three distinct encryption mechanisms to ensure the confidentiality of the attestation information, to comply with $\mathbf{R}_4$. First, we opted to use the recently-introduced encrypted client hello (ECH) for TLS [399], which is currently an IETF draft to communicate early information while preserving its privacy in the *ClientHello* message. To the best of our knowledge, we are the first to leverage this draft to protect the attestation request (**AttReq**) in the TLS handshake. In a nutshell, the *ClientHello* message is split into two parts: the outer message, which is in plain text, and the inner message, which is encrypted using a public key, typically distributed using the DNS infrastructure. Second, we rely on the TLS 1.3 encrypted extension in the broker reply (*i.e.*, the `ServerHello` message), so the broker's evidence remains confidential. This has been made possible because the two ends of the communication channel can derive a shared secret for encryption up to this point. Finally, the peer's certificate, containing the peer's evidence, is also protected, as the entire third handshake message is encrypted by design.

8.5.2 Securing publish/subscribe systems

We leverage TEEs and trusted Wasm runtimes for implementing our pub/sub design, ensuring strong hardware isolation of both code and data. This design remains versatile, working with various TEE architectures as long as they offer the trusted primitives for the Wasm runtimes (as detailed in Section 8.4.3). When paired with the mutually attested TLS protocol, we shield data in use and in transit, satisfying $\mathbf{R}_5$. As a pub/sub software, we selected Mosquitto [367], a well-known and open-source message broker that implements the latest version of the MQTT protocol. We have chosen Mosquitto because it is lightweight and suitable for use on all devices, from low-power IoT devices to cloud servers. Besides, we used WolfSSL, an embeddable cryptographic library, enabling the host of the TLS termination directly in the TEE, so external systems cannot eavesdrop on the communication, nor alter the code that maintains the endpoint. Minor modifications were made to the Mosquitto source code to compile it in Wasm, while the WolfSSL library was reused from Twine in Chapter 6, which had already been compiled in Wasm.

Mosquitto performs system calls for interactions with the outside world, such as using socket API for exposing TLS endpoints. For such usages, we rely on WebAssembly System Interface (WASI), which translates the system calls to the underlying OS transparently. As the private and session keys of TLS are located within the TEE, the communication remains confidential against eavesdropping attempts, even when system calls are monitored. Besides, Mosquitto can persist undelivered messages in a database for later transmission to offline subscribers. WASI includes a file system API for data storage, while TEEs typically provide means to save files via a trusted API securely. As such, we rely on the Intel Protected File System (IPFS) integrated with TWINE in Chapter 6 to encrypt files, ensuring the confidentiality of the stored messages.

Our proposal ensures broker and peer trustworthiness through mutual attestation, inherently restricting adversaries from propagating and observing messages in the pub/sub system. To further reduce peer scope, we propose using access control lists (ACLs), which is a built-in functionality of Mosquitto. When paired with the X.509 certificates presented in the TLS handshake, this feature can authenticate peers without requiring additional usernames or passwords, as this is typically the case using Mosquitto. This narrowed publication and subscription scope satisfies $\mathbf{R}_6$. Future work may further adapt the ACLs to be bound to evidence, provided that code measurements differ from each actor.

Our approach applies to various distributed pub/sub systems, enabling scalable communication through publisher-subscriber decoupling and secure messaging via mutual attestation. This was demonstrated in our European-funded project with Siemens for secure edge nodes data processing and capturing results on an MQTT messaging bus, and with the Byzantine fault-tolerant (BFT) system of the University of Lisbon for maintaining trust in the attestation reference values [400]. To balance the need for compactness with computational performance, we used Intel NUCs as edge nodes, which are small, efficient PCs equipped with processors that support Intel SGX. This allows the publisher on the edge nodes to verify whether the broker is genuine based on evidence and the code hash contained in that payload. This code hash is then compared to the trusted reference values stored in the BFT-resilient system. The evaluation (in Section 8.6) details how the broker performs when handling heavy loads from multiple publishers and subscribers, as encountered in these distributed environments.

8.5.3 Implementation

We developed a prototype of a trusted pub/sub broker using Intel SGX to better understand the practical impacts and performance overhead associated with our proposal. As such, we used TWINE (detailed in Chapter 6), a trusted Wasm runtime designed to secure applications on Intel SGX. We target the latest Intel SGX implementation (Intel Scalable) for creating enclaves with an Enclave Page Cache (EPC) up to 512 GiB, whereas prior work was limited to 128 MiB, which was a performance bottleneck when reaching that threshold.

Generating attestation evidence We provided the two attestation primitives, *i.e.*, *generate* and *verify* (defined in Section 8.4.3), which use the Intel SGX attestation feature. The former primitive calls the underlying TEE API to create a payload in the JSON format, which is later communicated in the TLS handshake. That payload notably contains the type of TEE and the

evidence itself. The latter primitive receives a JSON and indicates whether the remote system is genuine. The runtime abstracts these primitives using *librats* [137], a library capable of generating and verifying attestation of many TEE implementations. For each TEE implementation, librats includes a corresponding verifier for appraising evidence, decoupling the TEE architectures concerns from the Wasm applications. Evidence issued from TWINE includes assertions to ensure the TCB is up-to-date, and code measurements of the runtime and the hosted Wasm application, along with the keying materials of the current TLS session. Evidence allows the other actor (*i.e.*, the publisher, subscriber or broker) to attest the enclave. As a remote attestation mechanism, we use the Intel Data Center Attestation Primitives (DCAP) (covered in Section 3.4.3), which avoids involving Intel during the verification of evidence, speeding up the process and allows our pub/sub system to operate without an Internet connection.

Compiling into Wasm We compiled Mosquitto using Clang with WASI-SDK [340], a toolchain that compiles C/C++ source code into Wasm for non-web environments. Mosquitto lacks support for a portable and Wasm-enabled TLS library. Therefore, we modified Mosquitto to use WolfSSL as a drop-in cryptographic library replacement. We leverage the compatibility layer of WolfSSL with OpenSSL, minimising henceforth the required code changes. Further, we disabled some of Mosquitto's system interaction layers (*e.g.*, signals, dynamic library loading) as not supported by WASI. Turning off these features allowed a smooth embedding of Mosquitto for Wasm into the SGX enclave. The adaptation of Mosquitto required changes to 610 SLOC (in C source/header files), which accounts for 1.2 % of the total codebase.

Securing the file system Our proposal relies on the trusted file system of TWINE, which is backed by the Intel Protected File System (IPFS). IPFS uses AES-GCM for encryption, taking advantage of hardware acceleration from the CPU. Files are encrypted into a Merkle tree structure of 4 KiB nodes and stored on the untrusted file system, with each node securing its children through encryption keys and tags. When the enclave requests data, IPFS decrypts tree nodes within the shielded memory of the TEE, ensuring the information stored on the untrusted file system remains confidential and unaltered. In our proof of concept, IPFS automatically generates the keys based on enclave signatures and processor-specific keys. For more sophisticated use cases, such as fault tolerance, Wasm programs can provide their encryption keys, which can be obtained from an external source like a key management system (KMS).

Supporting the cloud-edge continuum We opted for Intel SGX for hosting our proof of concept because TWINE supports all the trusted primitives outlined in Section Section 8.4.3, which are necessary for running our proposal. While alternative implementations are possible on other TEE architectures like Intel TDX and AMD SEV-SNP using the runtime WAMR [117], or Arm TrustZone using WATZ (detailed in Chapter 7), these options would require additional effort because they currently support only some of the trusted primitives our proposal relies on. For example, WATZ enhances Arm TrustZone with attestation functionalities but does not offer file persistence or a complete socket API. In the case of WAMR, its mechanisms for attestation have yet to be tested on AMD platforms. Nonetheless, we are confident that with some extensions to these runtimes, our prototype can be adapted to work with a variety of TEE architectures, enabling an agnostic approach for trustworthy execution of pub/sub systems.

8.6 Evaluation

Our evaluations assess several aspects of our proof of concept. We intend to answer the following questions: (i) What performance costs are associated with the use of Wasm, SGX, and the creation of attestation evidence on the broker during the connection process? (ii) How does our solution scale when increasing message throughput? (iii) How does our solution scale with a growing number of publishers? To answer these questions, we measure the connection times with different peers (§8.6.1), stress the broker with a high volume of messages (§8.6.2), and grow the number of publishers while observing the resulting latency (§8.6.3).

Our evaluation is deliberately centred on the broker, which is the core and critical part of distributed pub/sub architectures. As the broker orchestrates the message flow among publishers and subscribers, it largely influences the scalability of the system. We examine the broker in three variants: native execution without SGX (baseline), Wasm outside SGX, and Wasm-SGX running within SGX. To ensure a consistent comparison, publishers and subscribers are executed in their native environment without SGX. While our focus remains on the broker to evaluate performance benchmarks, a fully secure system requires that all entities operate within TEEs to protect the confidentiality of data, including publishers and subscribers.

The broker runs on a 16-core Intel Xeon Gold 6326 (2.9 GHz), running Ubuntu 20.04, SGX Scalable (EPC of 64 GiB) with SGX driver v1.41 (DCAP) and SDK v2.20. The publishers and subscribers are executed on an 8-core Intel Xeon E3-1270 v6 (3.8 GHz), running Ubuntu 20.04. These two machines are connected through a 1 Gbit/s switch. We modified Mosquitto v2.0.15 and WolfSSL v5.5.3, and compiled them using Clang v10 at maximum optimisation. For Wasm benchmarks, we compiled the application into Wasm bytecode with Clang, and used ahead-of-time (AOT) compilation into assembly code using WAMR's compiler at level size 1, as required for SGX. Time is measured using Linux's monotonic clock, including enclave exit and re-entry times (22 µs). The MQTT broker is configured to match the minimum quality of service (QoS) offered, *i.e.*, fire-and-forget mode.[1] Our implementation is open-source, and instructions to reproduce our experiments are available on a GitHub repository [16].

8.6.1 Establishing new connections

Our initial evaluation measures the average latency incurred when a varying number of peers attempt to establish new connections with a broker. In this setup, we instruct a certain number of peers (depicted on the x-axis) to start a new connection every second. The y-axis displays the resulting latency. Figure 8.3a depicts the latency for connections made using TLS handshakes with elliptic curve Diffie-Hellman (ECDH) as a key agreement protocol. Figure 8.3b shows the latency when using a pre-shared key (PSK), which avoids using asymmetric cryptography. The latter approach is applicable when a peer has previously established a connection with the broker and already has a mutually agreed-upon secret key.

[1] MQTT supports two additional QoS levels: at least once with confirmation required and delivering it exactly once. Our protocol avoids the overheads of a four-step handshake while ensuring peer integrity and authenticity.

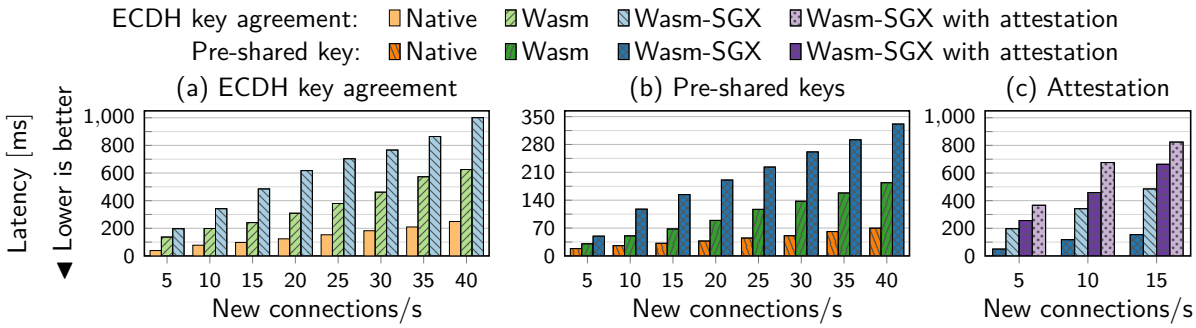


Figure 8.3: The latency for each new connection at varying connections per second.

Across both (ECDH and PSK) scenarios, latency scales linearly for every variant as the number of peers connecting per second increases from 5 to 40. We observe higher latencies with Wasm (using ECDH) over native execution with an overhead of 2.57×, due to Wasm's performance constraints relative to native execution [116]. Additionally, Wasm-SGX (using ECDH) exhibits further overhead (4.72×) compared to native because of the enclave, which is expected given the added security benefits. A contributing source of this overhead is the frequent switching between the enclave's secure and standard execution modes (OCALLs). This is particularly evident during operations that initialise sockets and manage peer communication in the TLS handshake. For instance, using a non-blocking socket API requires constant polling when awaiting client responses. Although such overhead is also present without SGX, it is accentuated due to the enclave transitions. We point out that the data remains confidential upon leaving the enclave, as the TLS protocol operates within a secure environment. More generally, SGX also performs slower than standard execution due to security mechanisms introduced in the microcode [401].

Figure 8.3c explores the impact of integrating broker attestation evidence in the TLS handshake. Wasm-SGX (using ECDH) with attestation incurs a 1.87× overhead, while Wasm-SGX (using PSK) exhibits a 4.33× overhead compared to their non-attested counterparts. Notably, the system saturates when exceeding 15 new ECDH connections per second, whereas PSK performance remains stable. The additional latency is primarily caused by the asymmetric operations involved in evidence generation and signing. Mosquitto's *broker bridge* feature can distribute the workload across multiple brokers to improve scalability and manage this overhead over a larger peer set, although an in-depth analysis is left for future research.

8.6.2 Messages throughput

Figure 8.4a evaluates the system's throughput by measuring the average latency of delivering a set number of messages per second (x-axis) with a single publisher and subscriber through a specific broker. The resulting latencies are reported on the y-axis. Each test case spans 60 seconds and includes a 16 KiB payload of random data.

Latency for the native variant slightly increases at 50 messages per second, followed by a minor decline at 200 messages per second. In contrast, we note a more pronounced decrease in latency for the Wasm and Wasm-SGX variants as the rate of publications increases. To understand this

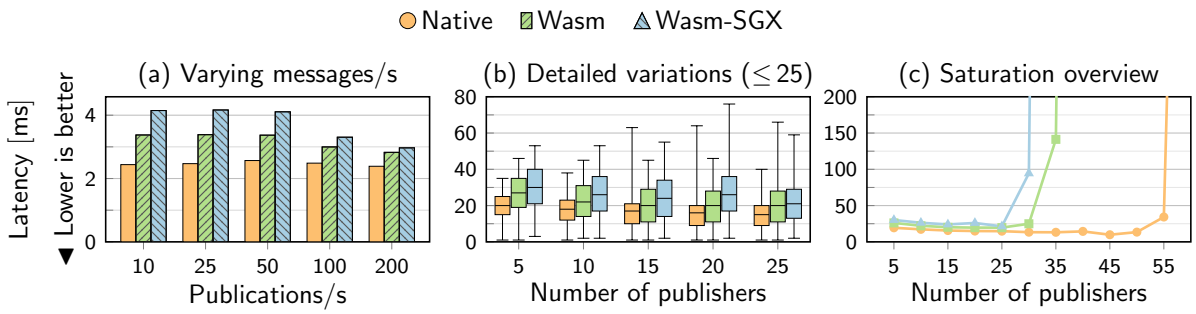


Figure 8.4: (a) shows latency with varying messages per second, (b) and (c) depict latency as publisher count grows.

behaviour, we instrumented the Wasm runtime to monitor the enclave's outgoing calls (OCALLs) and analysed the invoked POSIX functions. Enclave transitions typically incur performance costs, such as the intrinsic SGX transition and the copy of message buffers into secure memory. We discovered that POSIX functions related to network message transactions (*i.e.*, `recvfrom` and `send`) were called less frequently as the messages-per-second rate increased. For instance, at a rate of 10 messages per second, the frequencies of `recvfrom` and `send` are 2601 and 1411, respectively, amounting to an average of 260 and 141 per individual message. Conversely, at a higher rate of 200 messages per second, these frequencies adjust to 49 156 and 23 498, respectively, meaning a smaller average of 246 and 117 per message. These observations likely originate from Mosquitto's broker behaviour and were not investigated further.

Overall, the Wasm and Wasm-SGX variants show performance slowdown factors of 1.30× and 1.55×, respectively, compared to native execution. However, the system exhibits strong scalability and offers enhanced security and portability benefits, offsetting the performance impact.

8.6.3 Scaling the publishers

As our final experiment, we assess system scalability by increasing the number of publishers plotted on the x-axis, to observe the impact on message delivery latency, shown on the y-axis. The number of subscribers is held constant at 25, with each publisher sending 5 messages per second. Each test case lasts 60 seconds and uses a payload of 16 KiB of random data.

Figure 8.4b reveals that all the variants follow a similar trend. Latency generally remains stable but decreases slightly as the number of publishers increases. The slowdown of Wasm compared to native execution is 1.31×, while for Wasm-SGX against native variant is 1.56×.

Our analysis further investigates the point at which the system begins to saturate, thereby affecting its responsiveness. Figure 8.4c highlights this limit for each variant, indicating system saturation when exceeding 25 publishers for Wasm-SGX, followed by Wasm at 30 publishers. The native variant only reaches its limit beyond 55 publishers. Similarly to the first experiment, we could use Mosquitto's broker bridge feature to handle more publishers. This would increase the overall capacity and reduce the load of individual brokers.

8.7 Synthesis

Recent evolution in TEEs by leading CPU manufacturers highlights the growing trend in executing software within untrusted environments while processing increasingly sensitive data. The pub/sub model stands out as an effective mechanism to scale and distribute computations across varied architectures, and Wasm emerges as a suitable common environment for such tasks. However, a gap remains in establishing standardised protocols and tools that leverage the rapid research breakthroughs in trusted execution within the cloud-edge continuum.

To address this gap, the trusted Wasm runtimes presented in Chapters 6 and 7 can be used to span the cloud-edge continuum and provide a common communication layer that ensures the trustworthiness of distributed peers. This secure communication is achieved by extending the well-established TLS protocol to include mutual attestation and embed evidence through TLS handshake extension points, while maintaining compatibility with the original standard.

The evaluation results suggest that the security and portability enhancements of this approach effectively balance the additional overheads, which are primarily associated with the TEEs. Furthermore, leveraging Mosquitto's functionalities to distribute brokers among peers can further optimise the system, leading to a scalable and secure architecture suitable for large-scale and real-world applications. The implementation is openly available as an open-source project [16].

This chapter is the final building block of this thesis and leverages Twine and WaTZ in heterogeneous environments, using Intel SGX and Arm TrustZone, respectively. Recent technological releases from CPU manufacturers have shown a trend towards aligning with the coarse-grained trust boundary of VM-based TEEs, as demonstrated by Intel TDX [107] and Arm CCA [108, 153]. Wasm is also well-suited for execution within this trust boundary, as the runtime is a regular process with access to attestation functionalities. Consequently, the proposal presented in this chapter can be reused as-is, thanks to the abstraction that Wasm offers.

The following and final chapter concludes this dissertation. In particular, it reviews on the contributions of this thesis and identifies additional research topics and challenges for enhancing the adoption of Wasm for standalone execution within trusted execution environments.

Chapter 9

Conclusion

This chapter summarises the contributions and takeaways presented in this dissertation, discussing promising open research perspectives along with possible future directions at the intersection of WebAssembly (Wasm), trusted execution environments (TEEs) and attestation.

Chapter outline

9.1 Thesis summary and takeaways 156
9.2 Research perspectives and future directions 158
9.2.1 Better WebAssembly compatibility with legacy applications 158
9.2.2 Attestable and offloadable computations 159
9.2.3 Widening support for emerging trusted execution environments 160
9.2.4 Standardisation of attestation primitives and protocols 161
9.2.5 Live migration of WebAssembly applications 161
9.3 Concluding remarks 162

Confidential computing has emerged as an increasingly important trend for IoT, edge and cloud environments, materialised through the adoption of trusted execution environments (TEEs). While academia has extensively researched TEEs for over two decades, the industry has only recently started adopting them for cloud settings, commonly referred to as cloud confidential computing. Despite this recent focus, developing trusted applications (TAs) remains a challenging task, both conceptually and technologically. Furthermore, the paradigm introduced with cloud confidential computing is incompatible with the existing TEEs for IoT and edge devices, creating a fracture that prevents the implementation of a seamless cloud-edge continuum. This dissertation provides solutions that address the complexity of programming, deploying, and establishing trust in these complex trusted environments by leveraging WebAssembly (Wasm), a portable compilation target and binary instruction format. Wasm is designed to enable secure, efficient, and interoperable execution across various platforms, making it an ideal candidate for bridging the gap between cloud and edge and IoT confidential computing paradigms.

In the remainder of this dissertation, I outline the main takeaways, findings, and contributions of my work in Section 9.1. Afterwards, in Section 9.2, I explore research perspectives and potential future directions. Finally, I conclude with some final remarks in Section 9.3.

9.1 Thesis summary and takeaways

This dissertation started contextualising the research landscape about trusted execution and its many technologies, considering the various environment types, among IoT, edge and cloud settings, leading to the concrete solution of TEEs. Portable compilation targets were also introduced alongside Wasm, highlighting the benefits of this novel software abstraction standard and why it is well-suited for running within TEEs. These topics were covered in detail in Chapter 1, setting the foundation for the rest of the dissertation. Besides, research challenges were elaborated and summarised as two main pillars: (i) the abstraction of the complexity for programming TAs for TEEs, and (ii) how such an abstraction can be leveraged to infer trust into distrusted parties. Such pillars were further studied in the corresponding manuscript parts.

Part I: Background

Part I laid the groundwork for the three essential components used in this research: trusted execution environments (TEEs), attestation, and WebAssembly (Wasm). This part strove to provide readers with a comprehensive understanding of the core concepts and offered a broad view of the current research fields, serving as the foundation for the following parts of the thesis.

Chapter 2 reviewed modern TEEs, abstracting the different trust boundaries used by the existing solutions and presenting state-of-the-art implementations along with an in-depth comparison of their security variations and limitations. This thesis addressed some of these shortcomings by leveraging Wasm, such as the restricted number of programming languages supported by the official SDKs. Besides, Chapter 3 explored how attestation ensures the genuineness of trusted environments, explaining attestation agnostic concepts and surveying how Intel and AMD TEE implementations guarantee trustworthy code execution. This chapter was instrumental in designing an attestation protocol in Chapter 7, which is a missing feature for Arm TrustZone. The

combination of these two chapters led to a workshop paper presented at SysTEX'22 [7], which was further expanded into a conference paper published in DAIS'22 [6].

Chapter 4 examined the fundamentals of Wasm and its potential as an interoperable solution across the cloud-edge continuum. It also highlighted the key benefits of Wasm, resulting from its abstraction of the programming languages, the system environment, the execution modes, and the operating system services. These abstractions enabled Wasm to offer a more flexible, efficient, and secure runtime environment. The chapter then explored how these advantages manifest when integrating Wasm within TEEs. It concluded with a survey of Wasm runtimes, which guided the selection of the most suitable runtime for the implementation of TWINE and WATZ in Chapters 6 and 7, respectively. WAMR emerged as the optimal choice, offering a balance of flexibility and performance that aligns with the requirements of the TEE implementations targeted by this dissertation. Parts of this chapter were published as a workshop paper at FRAME'22 [5], promoting the benefits of Wasm and TEEs for the cloud-edge continuum.

Part II: WebAssembly as an efficient and attestable abstraction

Part II leveraged Wasm for developing portable applications, with a dual focus on enhancing web performance and securing trusted applications across various TEE implementations. The research presented in this part demonstrated that Wasm is well-suited for a variety of constrained environments as a platform-agnostic solution with its memory-safe sandboxed execution environment and compact binary format while achieving near-native performance.

Chapter 5 dived into web development techniques that offload cryptographic operations to the client-side browser, reducing the computational burden on server resources. Rather than depending on JavaScript implementations or restricted browser capabilities, this chapter demonstrated the compilation of the WolfSSL library using Wasm as a portable compilation target. This approach supported a wide range of cryptographic primitives and exhibited significant performance improvements compared to JavaScript-based cryptographic libraries. The research, conducted in collaboration with Swiss Post, aimed to enhance their secure email service, IncaMail. The findings from this chapter were presented at the DEBS'22 conference [3].

Chapter 6 presented TWINE, a trusted runtime for Intel SGX that hosts entire applications within the enclave, diverging from the traditional approach of splitting software into untrusted and trusted parts. TWINE uses WebAssembly System Interface (WASI) as a system interface, mapping POSIX-like system calls to trusted SGX primitives or securely relying on the untrusted OS while preventing the leakage of confidential information. Notably, TWINE protects file system interactions using Intel Protected File System (IPFS), which transparently encrypts files from the perspective of the hosted application. Furthermore, TWINE integrates Intel SGX's attestation service, enabling Wasm-compiled applications to establish secure and attestable communication channels with remote endpoints. The fintech company *Credora Inc.* successfully adopted TWINE in its production infrastructure to protect critical assets. The research presented in this chapter was published as a conference paper at ICDE'21 [8] and extended as a journal article in the IEEE Transactions on Dependable and Secure Computing [1].

Chapter 7 introduced WATZ, a trusted runtime for Arm TrustZone that leverages OP-TEE as an open-source trusted OS. The Wasm-compiled applications are hosted within a user space TA and

executed in the secure world. Akin to TWINE, WATZ uses WASI to abstract the implementation details of the system interface exposed by the GlobalPlatform (GP) API. To address the absence of built-in attestation primitives in this TEE implementation, WATZ includes a remote attestation mechanism. This is achieved through a dedicated kernel module for OP-TEE, designed to issue evidence based on a root of trust that is exclusively accessible from the kernel space of the secure world. The research on WATZ was published as a conference paper at ICDCS'22 [4].

The common execution of Wasm applications across these disparate TEE paradigms shows the feasibility of establishing a common ground for software execution despite significant differences in their implementations. TWINE and WATZ demonstrate the ability to run legacy programs compiled into Wasm with minimal performance overheads while simultaneously providing trust primitives and assurance through the use of trusted API and attestation mechanisms.

Part III: Establishing trust using WebAssembly for network communication

Part III built upon the trusted Wasm runtimes and their attestation primitives to establish secure communication channels between mutually distrusted remote parties.

Chapter 8 leveraged the MQTT messaging protocol and the TLS security protocol to carry attestation evidence and facilitate mutual attestation of remote peers running in different TEE implementations. This additional abstraction layer compiled using Wasm and built on top of these communication standards provides a portable attestation mechanism that addresses the lack of standardisation in transporting attestation data. As a result, applications running within these trusted runtimes can transparently attest to one another before initiating communication, as they are not tied to a specific TEE implementation, programming language, or operating system. This work was accepted for publication as a conference paper at OPODIS'23 [2].

Lastly, the research presented in this thesis yielded several software artefacts (listed in Section 1.8). Some of these artefacts were merged back into their original repositories to promote reproducibility and encourage broader industry adoption. For instance, most of the TWINE functionalities were merged into the official GitHub repository of WAMR.

9.2 Research perspectives and future directions

The results of this thesis motivate new research directions for TEEs, Wasm, and attestation. The following sections outline potential future work building on the contributions of this work.

9.2.1 Better WebAssembly compatibility with legacy applications

The introduction of WASI 0.1 marked a major step forward in supporting the compilation of legacy applications to Wasm. This system service interface allowed for the compilation of complex software, such as SQLite, to run within TEEs with minimal modifications. However, despite this progress, WASI 0.1 still lacks support for certain system services that are surprisingly prevalent among server applications. For instance, Wasm-compiled applications with WASI capabilities are limited to operating on pre-opened sockets. To address these missing capabilities,

several Wasm runtimes, including WAMR, Wasmer, and WasmEdge, have proposed ad-hoc extensions to expose these system services to Wasm programs.

The industry partially addressed the missing features in Wasm by releasing WASI 0.2, a redesigned specification adopting a component model that facilitates extensibility for new requirements. However, all the interfaces of WASI 0.2 are currently in the implementation phase (*phase 3*) at the time of writing, indicating that these features are not yet fully standardised (*phase 5*), potentially introducing breaking changes in the future.

Wasm runtimes like Wasmer also proposed standards such as WASIX [402], which extends WASI 0.1 to deliver core functionalities for running real-world Wasm applications by better aligning with POSIX standards. WASIX includes features like efficient multithreading, sockets, current directory support, subprocess spawning, and console support. Nevertheless, the extensive API support required by WASIX can hinder its adoption in constrained environments like TEEs due to the significant upfront effort needed to support all features, which may not be readily available.

Academia has also contributed to addressing these limitations through proposals like WALI [403], a thin layer over Linux's userspace system calls that enables Wasm to interact with native processes and the underlying OS. However, this approach creates friction as it compromises two fundamental principles of Wasm: the OS-neutral approach, as it relies on Linux's system calls, and the capability-based security model, which is crucial for enforcing resource sandboxing.

Designing a pragmatic yet production-ready solution for supporting legacy software system dependencies is a challenging task. As prior work has shown, this requires striking a balance between lightweight standards and full-featured specifications, with the risk of introducing yet another competing standard that could fragment the ecosystem [404]. The modular approach introduced in WASI 0.2 addresses this challenge by enabling the composition of dependencies. This design breaks down potentially large future API standards into compact, manageable chunks that can be partially supported in constrained environments. While this represents a significant step forward, further work remains to be done to improve these specifications. One area for future research is the official proposal of a WASI standard for attestation, which would enhance the security and trustworthiness of WASI-based systems deployed in TEEs.

9.2.2 Attestable and offloadable computations

This dissertation focused on CPU-bound TEEs, which are trusted environments instantiated by an extension of the CPU's instruction set architecture (ISA). As a result, computations within these environments are limited to the processor's functionalities. However, many operations in cloud or on-premises require other device types, such as FPGAs and GPUs, for tasks like machine learning. These accelerators are typically considered outside the trusted computing base (TCB) of CPU-bound TEEs, limiting their use because confidential data, such as machine learning models, cannot be securely transferred to these devices with equivalent security guarantees.

Researchers have explored various solutions for ensuring data confidentiality when using accelerators (*e.g.*, GPUs, FPGAs) in conjunction with TEEs. These approaches include: (i) shielding device access using an exclusive path using TEE primitives, hardware primitives, or privileged software execution [405–407], (ii) partitioning techniques that split data and offload

non-sensitive parts to the accelerator, such as partitioning deep neural network (DNN) models [408–410], and (iii) obfuscating or encrypting confidential data in TEEs before delivering it to accelerators [411–413]. Nonetheless, these solutions often rely on ad-hoc implementations or face important drawbacks, such as a weaker threat model that does not exclude the hypervisor, or lack runtime integrity, require dedicated hardware, or incur performance overheads.

More recently, manufacturers have implemented TEEs in GPUs, such as NVIDIA Hopper architecture [414, 415], addressing the growing demand for confidential computing on accelerators. These TEEs shield user data from hardware and software attacks while better isolating and protecting virtual machines (VMs) assets from each other in virtualised environments. This isolation is achieved using multi-instance GPU (MIG), which partitions the GPU into smaller, isolated TEE instances with dedicated resources, supporting multi-tenant cloud configurations. Designed to integrate with Intel TDX and AMD SEV, these two VM-based TEEs establish encrypted communication channels and attest genuine hardware and firmware before operation.

Wasm, as a virtual ISA, lacks direct hardware interaction, preventing the use of hardware accelerations like cryptographic operations, which are implemented as dedicated assembly instructions in most CPUs. To address this issue, recent initiatives such as *wasi-crypto* [270] and *wasi-nn* [253] extend WASI 0.2 by integrating modules for cryptography and machine learning, respectively, moving such capabilities to the runtime. By doing so, the runtime can then significantly improve Wasm application performance by offloading computations to hardware accelerators. These extensions create an indirection layer between the Wasm application and optimised software or hardware, abstracting the implementation based on the runtime's execution environment. Looking ahead, a promising direction for future research is the exploration of transparent offloadable computations to third-party TEEs, shifting the complexity of establishing trustworthy channels into the runtime and simplifying Wasm application development.

9.2.3 Widening support for emerging trusted execution environments

The swift evolution of confidential computing led to new TEE implementations, such as Intel TDX [107] and Arm CCA [108]. These approaches align with the existing TEE implementation AMD SEV [106], which emphasises the VM-based paradigm. While these technologies present a larger attack surface due to their increased TCB size, they represent the future of cloud confidential computing, as Intel, AMD, and Arm are the leading CPU vendors. Supporting these TEEs will pave the way for a better cloud-edge continuum, extending current TEE solutions.

RISC-V, an open standard and royalty-free ISA, has also designed extensions to support TEEs. The Physical Memory Protection (PMP) instructions [195] were the first step, enabling software to configure the memory protection unit to shield memory based on privilege levels (*i.e.*, rings). While some TEE implementations listed in Section 2.2.4 use PMP instructions for enforcing isolation, these low-level primitives require accessing higher privilege levels for effective isolation, preventing the kernel and hypervisor from tampering with the protection. To offer a better abstraction, a more advanced VM-based TEE implementation named Confidential VM Extension (CoVE) [197, 198] was designed for RISC-V. CoVE adopts an approach similar to AMD SEV, Intel TDX, and Arm CCA, providing a higher abstraction over the PMP instructions. Although CoVE is still in the process of being ratified, securing the execution of Wasm applications

in such environments would offer valuable experience and insights, representing a significant step forward in confidential computing for the RISC-V architecture.

9.2.4 Standardisation of attestation primitives and protocols

TEE implementations use ad-hoc attestation mechanisms to trust TCB instances, hindering the adoption of a common approach for issuing and verifying attestation evidence. TWINE and WATZ partially addressed this strong coupling by offering TEE-neutral attestation primitives, which is simplified with Wasm due to its cross-platform binary compatibility. However, the standardisation of such attestation primitives remains absent. WATZ drafted *WASI-RA* as a candidate for a loosely coupled abstraction between Wasm applications and TEE implementations, with the runtime mediating and offering reusable data structures and API across platforms. Further efforts are required to establish a standard for this abstraction, and the new component model introduced by WASI 0.2 presents an opportunity for extending the specifications.

Attestation protocols define how attestation evidence is conveyed between parties and are typically coupled with transport protocols like TLS to authenticate peers during communication based on both evidence and certificates. Chapter 8 proposed carrying evidence through TLS 1.3 by leveraging its new encryption extensions while preserving certificates signed by global certificate authorities (CAs). Although this design offers several advantages, it represents another fixed design point in a solution for composing attestation and transport protocols. Efforts must be directed towards the standardisation of such protocols to facilitate the implementation of frameworks and tools targeting the cloud-edge continuum [138]. Current initiatives, such as an IETF Internet-Draft [144], are underway to propose standardised extensions for binding of the TLS authentication key to a remote attestation session. Participation in the design or implementation of these standardisation efforts would contribute to their broader adoption.

9.2.5 Live migration of WebAssembly applications

Live migration involves transferring a running program instance from one machine to another, akin to how hypervisors move executing VMs between servers using technologies like VMware vMotion [416]. Extensive research has been conducted on live VM migration for cloud and edge computing [417–419]. Besides, live migration of TEEs requires additional considerations and has been explored in several academic studies [420–424].

Applying live migration to Wasm applications can be motivated by various factors, such as (i) meeting strict latency constraints by migrating closer to mobile users, (ii) temporarily requiring high processing power and migrating to powerful cloud processors, (iii) balancing computational load across multiple devices, or (iv) moving closer to data collection points due to legal regulations or the need to process private data locally. However, the heterogeneous nature of the cloud-edge continuum poses significant challenges for live migration. Despite these challenges, the aforementioned use cases highlight the strengths of the continuum, which serves as a unified layer for software execution regardless of the type and location of devices.

Prior work studied the live migration of Wasm applications loaded in browsers by replicating the linear memory [425]. More recently, Nomad [426] and Fujii *et al*. [427] focused on migrating Wasm applications running in standalone runtimes. The former work lays the foundation of Wasm migrations, while the latter drafted a solution to migrate Wasm applications across different runtimes. When featuring TEEs, Edgedancer [428] studied the security implications and performance of live migration and of Wasm applications within Intel SGX enclaves. Pop *et al*. [429] further complements trusted live migration by detailing the security protocol for the migration process across different TEE implementations (Intel-based and Arm-based).

Despite the active research on live migration of Wasm applications shielded by TEEs, these solutions have yet to address the migration of resources bound to WASI. Real-world applications rely on system calls to operate, such as the file system for database engines and sockets for web servers. WASI functions can be categorised into two types: (i) stateless functions, which are self-contained and do not require any follow-up to operate or tear down, and (ii) stateful functions, which often require setting up a resource (like a file descriptor) and performing operations that change a state (the file cursor) maintained by the Wasm runtime. Stateful operations pose the primary challenges when migrating running applications, as serialising such resources is not trivial and is usually bound to a particular runtime or operating system. Migration of running processes is orthogonal to Wasm and has its own research techniques featuring *checkpoint-restart* techniques [430–432]. WASI can facilitate the implementation of such solutions by serving as a hooking mechanism for low-level OS API between the Wasm applications and the OS, potentially enabling OS-neutral capabilities. Implementing checkpoint-restart mechanisms directly in the WASI implementation, rather than hooking the user-space process or working as a kernel module, can provide a more efficient and portable solution.

9.3 Concluding remarks

In this dissertation, I have extensively studied trusted execution environments, attestation, and how WebAssembly can leverage these security principles to harden trusted applications while easing the development process and portability across many CPU manufacturers. The widespread adoption of many TEE implementations by leading CPU vendors demonstrates that confidential computing is set to become a long-lasting technology rather than a temporary security trend. Despite challenges such as side-channel attacks and the current lack of standardisation, academia and industry are converging on common practices, facilitating broader adoption of these technologies, including more seamless interoperability for the cloud-edge continuum. A prime example of this convergence is the widespread adoption of the VM-based TEE paradigm, such as Intel TDX, AMD SEV, Arm CCA, and RISC-V CoVE for cloud confidential computing. Moreover, recent research has revealed that existing VM-based TEE implementations exhibit superior performance for I/O and memory-intensive workloads compared to Intel SGX, a process-based TEE implementation [65]. However, it is important to note that process-based TEEs provide a stricter threat model, reducing the TCB size and, consequently, the attack surface, which maintains their viability as a compelling approach.

WebAssembly has proven to be an efficient and versatile open standard for standalone execution, including when hosted in TEEs. The widespread adoption of Wasm is exemplified by Docker, the leading platform for containerised applications, which recognises Wasm as a pivotal technology and now supports the execution of Wasm workloads. When Wasm applications are executed in Intel SGX and Arm TrustZone, as demonstrated by TWINE and WATZ, respectively, the incurred overheads are minimal and are outweighed by the benefits of ease of programming, deployment and enhanced software portability. The numerous open-source runtimes for standalone Wasm execution, which have been continuously maintained for several years, illustrate the strong commitment to supporting this technology and the inherent competitiveness of these alternatives drives the development of new features. Besides, WASI is the cornerstone standard for standalone execution, serving as a bridge between Wasm applications and operating systems. By focusing on a modular component model, the Wasm community can incrementally enhance the standard, avoiding the pitfalls of bloated execution environments.

As an engineer and aspiring researcher, I am excited to see how hardware manufacturers will enhance confidential computing standards with better cohesive interactions across implementations, and whether a more integrated cloud-edge continuum emerges with Wasm. The future holds many possibilities, and I look forward to being part of this computing journey.

□

Chapter 10

References

[1] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, Valerio Schiavoni, Giovanni Mazzeo, Arne Hollum and Darshan Vaydia. 2024. A comprehensive trusted runtime for WebAssembly with Intel SGX. *IEEE Transactions on Dependable and Secure Computing*, 21, 4, 3562–3579. DOI: 10.1109/TDSC.2023.3334516 (cit. on pp. 12, 157).

[2] Jämes Ménétrey, Aeneas Grüter, Peterson Yuhala, Julius Oeftiger, Pascal Felber, Marcelo Pasin and Valerio Schiavoni. 2023. A holistic approach for trustworthy distributed systems with WebAssembly and TEEs. In *27th International Conference on Principles of Distributed Systems, OPODIS 2023, December 6-8, 2023, Tokyo, Japan* (LIPIcs). Vol. 286. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 23:1–23:23. DOI: 10.4230/LIPICS.OPODIS.2023.23 (cit. on pp. 10, 12, 158).

[3] Pascal Gerig, Jämes Ménétrey, Baptiste Lanoix, Florian Stoller, Pascal Felber, Marcelo Pasin and Valerio Schiavoni. 2023. Preventing EFail attacks with client-side WebAssembly: the case of Swiss Post's IncaMail. In *Proceedings of the 17th ACM International Conference on Distributed and Event-based Systems, DEBS 2023, Neuchatel, Switzerland, June 27-30, 2023*. ACM, 151–156. DOI: 10.1145/3583678.3596899 (cit. on pp. 9, 12, 157).

[4] Jämes Ménétrey, Marcelo Pasin, Pascal Felber and Valerio Schiavoni. 2022. WaTZ: a trusted WebAssembly runtime environment with remote attestation for TrustZone. In *42nd IEEE International Conference on Distributed Computing Systems, ICDCS 2022, Bologna, Italy, July 10-13, 2022*. IEEE, 1177–1189. DOI: 10.1109/ICDCS54860.2022.00116 (cit. on pp. 10, 12, 158).

[5] Jämes Ménétrey, Marcelo Pasin, Pascal Felber and Valerio Schiavoni. 2022. WebAssembly as a common layer for the cloud-edge continuum. In *FRAME@HPDC 2022: Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge, Minneapolis, MN, USA, 1 July 2022*. ACM, 3–8. DOI: 10.1145/3526059.3533618 (cit. on pp. 9, 11, 124, 157).

[6] Jämes Ménétrey, Christian Göttel, Anum Khurshid, Marcelo Pasin, Pascal Felber, Valerio Schiavoni and Shahid Raza. 2022. Attestation mechanisms for trusted execution environments demystified. In *Distributed Applications and Interoperable Systems: 22nd IFIP WG 6.1 International Conference, DAIS 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings* (Lecture Notes in Computer Science). Vol. 13272. Springer, 95–113. DOI: 10.1007/978-3-031-16092-9_7 (cit. on pp. 9, 11, 157).

[7] Jämes Ménétrey, Christian Göttel, Marcelo Pasin, Pascal Felber and Valerio Schiavoni. 2022. An exploratory study of attestation mechanisms for trusted execution environments. In *5th Workshop on System Software for Trusted Execution, SysTEX 2022, co-located with ASPLOS'22, Lausanne, Switzerland, March 1, 2022*. https://systex22.github.io/papers/systex22-final79.pdf (cit. on pp. 9, 11, 157).

[8] Jämes Ménétrey, Marcelo Pasin, Pascal Felber and Valerio Schiavoni. 2021. Twine: an embedded trusted runtime for WebAssembly. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, 205–216. DOI: 10.1109/ICDE51399.2021.00025 (cit. on pp. 10, 12, 157).

[9] Louis Vialar, Jämes Ménétrey, Valerio Schiavoni and Pascal Felber. 2024. BlindexTEE: a blind index approach towards TEE-supported end-to-end encrypted DBMS. In *Stabilization, Safety, and Security of Distributed Systems - 26th International Symposium, SSS 2024, Nagoya, Aichi, Japan, October 20-22, 2024, Proceedings* (Lecture Notes in Computer Science). Springer.

[10] Mpoki Mwaisela, Joel Hari, Peterson Yuhala, Jämes Ménétrey, Pascal Felber and Valerio Schiavoni. 2024. Evaluating the potential of in-memory processing to accelerate homomorphic encryption. In *43rd International Symposium on Reliable Distributed Systems, SRDS 2024, Charlotte, USA, September 30–October 03, 2023*. IEEE.

[11] Peterson Yuhala, Jämes Ménétrey, Pascal Felber, Marcelo Pasin and Valerio Schiavoni. 2024. Fortress: securing IoT peripherals with trusted execution environments. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing, SAC 2024, Avila, Spain, April 8-12, 2024*. ACM, 243–250. DOI: 10.1145/3605098.3635994.

[12] Kevin Mika, René Griessl, Nils Kucza, Florian Porrmann, Martin Kaiser, Lennart Tigges, Jens Hagemeyer, Pedro Trancoso, Muhammad Waqar Azhar, Fareed Qararyah, Stavroula Zouzoula, Jämes Ménétrey, Marcelo Pasin, Pascal Felber, Carina Marcus, Oliver Brunnegård, Olof Eriksson, Hans Salomonsson, Daniel Ödman, Andreas Ask, António Casimiro, Alysson Bessani, Tiago Carvalho, Karol Gugala, Piotr Zierhoffer, Grzegorz Latosinski, Marco Tassemeier, Mario Porrmann, Hans-Martin Heyn, Eric Knauss, Yufei Mao and Franz Meierhöfer. 2023. VEDLIoT: next generation accelerated AIoT systems and applications. In *Proceedings of the 20th ACM International Conference on Computing Frontiers, CF 2023, Bologna, Italy, May 9-11, 2023*. ACM, 291–296. DOI: 10.1145/3587135.3592175.

[13] Martin Kaiser, René Griessl, Nils Kucza, Carola Haumann, Lennart Tigges, Kevin Mika, Jens Hagemeyer, Florian Porrmann, Ulrich Rückert, Micha vor dem Berge, Stefan Krupop, Mario Porrmann, Marco Tassemeier, Pedro Trancoso, Fareed Qararyah, Stavroula Zouzoula, António Casimiro, Alysson Neves Bessani, José Cecílio, Stefan Andersson, Oliver Brunnegård, Olof Eriksson, Roland Weiss, Franz Meierhöfer, Hans Salomonsson, Elaheh Malekzadeh, Daniel Ödman, Anum Khurshid, Pascal Felber, Marcelo Pasin, Valerio Schiavoni, Jämes Ménétrey, Karol Gugala, Piotr Zierhoffer, Eric Knauss and Hans-Martin Heyn. 2022. VEDLIoT: very efficient deep learning in IoT. In *2022 Design, Automation & Test in Europe Conference & Exhibition, DATE 2022, Antwerp, Belgium, March 14-23, 2022*. IEEE, 963–968. DOI: 10.23919/DATE54114.2022.9774653.

[14] Peterson Yuhala, Jämes Ménétrey, Pascal Felber, Valerio Schiavoni, Alain Tchana, Gaël Thomas, Hugo Guiroux and Jean-Pierre Lozi. 2021. Montsalvat: Intel SGX shielding for GraalVM native images. In *Middleware '21: 22nd International Middleware Conference, Québec City, Canada, December 6-10, 2021*. ACM, 352–364. DOI: 10.1145/3464298.3493406.

[15] [SW] Jämes Ménétrey, TWINE runtime and experiments 15 Jan. 2024. URL: https://github.com/jamesmenetrey/unine-twine (cit. on p. 96).

[16] [SW] Jämes Ménétrey, Aeneas Grüter and Julius Oeftiger, A Holistic Approach for Trustworthy Distributed Systems with WebAssembly and TEEs: code and benchmarks 1 Dec. 2023. URL: https://github.com/JamesMenetrey/unine-opodis2023 (cit. on pp. 150, 153).

[17] [SW] Jämes Ménétrey, WATZ runtime and experiments 19 May 2022. URL: https://github.com/jamesmenetrey/unine-watz (cit. on pp. 124, 134).

[18] [SW] Jämes Ménétrey, Clarify how to verify SGX evidence without an SGX-enabled platform 17 Feb. 2024. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/3158.

[19] [SW] Jämes Ménétrey, Attestation: free JSON from the Wasm module heap 22 Nov. 2023. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/2803.

[20] [SW] Jämes Ménétrey, SGX-RA: disable the building of samples 28 Aug. 2023. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/2507.

[21] [SW] Jämes Ménétrey, Upgrade SGX-RA integration for 0.1.2 and Ubuntu 20.04 15 Aug. 2023. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/2454.

[22] [SW] Jämes Ménétrey, Remove a file test outside of the specs and improve CI reporting 24 Mar. 2023. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/2057.

[23] [SW] Jämes Ménétrey, SGX IPFS: fix a segfault and support seeking beyond the end of files while using SEEK_CUR/SEEK_END 30 Jan. 2023. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1916.

[24] [SW] Jämes Ménétrey, linux-sgx: open files with any paths in the sandbox using IPFS 7 Nov. 2022. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1685.

[25] [SW] Jämes Ménétrey, linux-sgx: improve the documentation of SGX-RA 4 Nov. 2022. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1679.

[26] [SW] Jämes Ménétrey, linux-sgx: use non-destructive modes for opening files using SGX IPFS 27 Oct. 2022. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1645.

[27] [SW] Jämes Ménétrey, Normalize how the global heap size is defined across iwasm apps 25 Oct. 2022. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1628.

[28] [SW] Jämes Ménétrey, linux-sgx: implement POSIX calls based on getsockname and booloption 11 Oct. 2022. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1574.

[29] [SW] Jämes Ménétrey, linux-sgx: implement getpeername, recvfrom and sendto 6 Oct. 2022. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1556.

[30] [SW] Jämes Ménétrey, linux-sgx: fix directional OCALL parameter for getsockname 4 Oct. 2022. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1554.

[31] [SW] Jämes Ménétrey, Hash map: Fix a wrongly named parameter and enhance the docs 29 Sept. 2022. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1540.

[32] [SW] Jämes Ménétrey, Socket: Explicit narrowing type cast and add missing static keywords 29 Sept. 2022. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1539.

[33] [SW] Jämes Ménétrey, linux-sgx: Implement SGX IPFS as POSIX backend for files interaction 28 Sept. 2022. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1489.

[34] [SW] Jämes Ménétrey, Implement Berkeley Socket API for Intel SGX 25 Mar. 2022. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/1061.

[35] [SW] Jämes Ménétrey, Change whence_t constant values to match pre-existing agreed-upon values 10 July 2020. URL: https://github.com/bytecodealliance/wasm-micro-runtime/pull/307.

[36] [SW] Jämes Ménétrey, sgx_ecdsa: set the load_policy at most once 17 Nov. 2023. URL: https://github.com/inclavare-containers/librats/pull/94.

[37] [SW] Jämes Ménétrey, verifiers: use type instead of name for selection 14 Aug. 2023. URL: https://github.com/inclavare-containers/librats/pull/82.

[38] [SW] Jämes Ménétrey, Implement a TLS perf server with poll system call 27 Sept. 2022. URL: https://github.com/wolfSSL/wolfssl-examples/pull/330 (cit. on p. 100).

[39] [SW] Jämes Ménétrey, TLS client: fix bad comparison for non-blocking shutdown call 21 July 2022. URL: https://github.com/wolfSSL/wolfssl-examples/pull/327.

[40] [SW] Jämes Ménétrey, Update latex .bib file for citing WebAssembly specs 2.0 6 May 2022. URL: https://github.com/WebAssembly/spec/pull/1463.

[41] [SW] Jämes Ménétrey, imx: add a compilation flag to support the persistence of the rootfs 3 May 2021. URL: https://github.com/OP-TEE/build/pull/477.

[42] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andy Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica and Matei Zaharia. 2010. A view of cloud computing. *Commun. ACM*, 53, 4, 50–58. DOI: 10.1145/1721654.1721672 (cit. on p. 2).

[43] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li and Lanyu Xu. 2016. Edge computing: vision and challenges. *IEEE Internet Things J.*, 3, 5, 637–646. DOI: 10.1109/JIOT.2016.2579198 (cit. on p. 2).

[44] Hamed Tabrizchi and Marjan Kuchaki Rafsanjani. 2020. A survey on security challenges in cloud computing: issues, threats, and solutions. *J. Supercomput.*, 76, 12, 9493–9532. DOI: 10.1007/S11227-020-03213-1 (cit. on p. 2).

[45] Fatemeh Khoda Parast, Chandni Sindhav, Seema Nikam, Hadiseh Izadi Yekta, Kenneth B. Kent and Saqib Hakak. 2022. Cloud computing security: A survey of service-based models. *Comput. Secur.*, 114, 102580. DOI: 10.1016/J.COSE.2021.102580 (cit. on p. 2).

[46] Stefan Berger, Ramón Cáceres, Dimitrios E. Pendarakis, Reiner Sailer, Enriquillo Valdez, Ronald Perez, Wayne Schildhauer and Deepa Srinivasan. 2008. TVDc: managing security in the trusted virtual datacenter. *ACM SIGOPS Oper. Syst. Rev.*, 42, 1, 40–47. DOI: 10.1145/1341312.1341321 (cit. on p. 2).

[47] Mihai Christodorescu, Reiner Sailer, Douglas Lee Schales, Daniele Sgandurra and Diego Zamboni. 2009. Cloud security is not (just) virtualization security: a short paper. In *Proceedings of the first ACM Cloud Computing Security Workshop, CCSW 2009, Chicago, IL, USA, November 13, 2009*. ACM, 97–102. DOI: 10.1145/1655008.1655022 (cit. on p. 2).

[48] National Vulnerability Database. 2015. CVE-2015-2337, VMware. National Institute of Standards and Technology. (13 June 2015). Retrieved 5 June 2024 from https://nvd.nist.gov/vuln/detail/CVE-2015-2337 (cit. on p. 2).

[49] National Vulnerability Database. 2015. CVE-2015-3456, QEMU. National Institute of Standards and Technology. (13 May 2015). Retrieved 5 June 2024 from https://nvd.nist.gov/vuln/detail/CVE-2015-3456 (cit. on p. 2).

[50] National Vulnerability Database. 2015. CVE-2015-5154, Xen. National Institute of Standards and Technology. (12 Aug. 2015). Retrieved 5 June 2024 from https://nvd.nist.gov/vuln/detail/CVE-2015-5154 (cit. on p. 2).

[51] National Vulnerability Database. 2016. CVE-2016-0088, Microsoft Hyper-V. National Institute of Standards and Technology. (12 Apr. 2016). Retrieved 5 June 2024 from https://nvd.nist.gov/vuln/detail/CVE-2016-0088 (cit. on p. 2).

[52] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth and Berk Sunar. 2014. Fine grain cross-VM attacks on Xen and VMware. In *2014 IEEE Fourth International Conference on Big Data and Cloud Computing, BDCloud 2014, Sydney, Australia, December 3-5, 2014*. IEEE Computer Society, 737–744. DOI: 10.1109/BDCLOUD.2014.102 (cit. on p. 2).

[53] Atif Saeed, Peter Garraghan and Syed Asad Hussain. 2022. Cross-VM network channel attacks and countermeasures within cloud computing environments. *IEEE Trans. Dependable Secur. Comput.*, 19, 3, 1783–1794. DOI: 10.1109/TDSC.2020.3037022 (cit. on p. 2).

[54] Udaya Kiran Tupakula, Vijay Varadharajan and Dipankar Dutta. 2012. Intrusion detection techniques for virtual domains. In *19th International Conference on High Performance Computing, HiPC 2012, Pune, India, December 18-22, 2012*. IEEE Computer Society, 1–9. DOI: 10.1109/HIPC.2012.6507491 (cit. on p. 2).

[55] Zvika Brakerski, Craig Gentry and Vinod Vaikuntanathan. 2012. (leveled) fully homomorphic encryption without bootstrapping. In *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*. ACM, 309–325. DOI: 10.1145/2090236.2090262 (cit. on pp. 3, 20).

[56] Craig Gentry. 2009. *A fully homomorphic encryption scheme*. PhD thesis. Stanford University, USA. https://searchworks.stanford.edu/view/8493082 (cit. on pp. 3, 20).

[57] Paulo Martins, Leonel Sousa and Artur Mariano. 2018. A survey on fully homomorphic encryption: an engineering perspective. *ACM Comput. Surv.*, 50, 6, 83:1–83:33. DOI: 10.1145/3124441 (cit. on pp. 3, 20).

[58] Ran Canetti, Uriel Feige, Oded Goldreich and Moni Naor. 1996. Adaptively secure multi-party computation. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*. ACM, 639–648. DOI: 10.1145/237814.238015 (cit. on pp. 3, 20).

[59] Sherman S. M. Chow, Jie-Han Lee and Lakshminarayanan Subramanian. 2009. Two-party computation model for privacy-preserving queries over distributed databases. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009*. The Internet Society. https://www.ndss-symposium.org/ndss2009/two-party-computation-model-privacy-preserving-queries-over-distributed-databases/ (cit. on pp. 3, 20).

[60] Yang Yang, Xindi Huang, Ximeng Liu, Hongju Cheng, Jian Weng, Xiangyang Luo and Victor Chang. 2019. A comprehensive survey on secure outsourced computation and its applications. *IEEE Access*, 7, 159426–159465. DOI: 10.1109/ACCESS.2019.2949782 (cit. on pp. 3, 20).

[61] Juan Manuel González Nieto, Ed Dawson and Eiji Okamoto. 2003. Privacy and trusted computing. In *14th International Workshop on Database and Expert Systems Applications (DEXA'03), September 1-5, 2003, Prague, Czech Republic*. IEEE Computer Society, 383–388. DOI: 10.1109/DEXA.2003.1232052 (cit. on pp. 3, 20).

[62] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, André Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Peter R. Pietzuch and Christof Fetzer. 2016. SCONE: secure Linux containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 689–703. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov (cit. on pp. 3, 6, 8, 12, 97, 102).

[63] Chia-che Tsai, Donald E. Porter and Mona Vij. 2017. Graphene-SGX: a practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. USENIX Association, 645–658. https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai (cit. on pp. 3, 6, 8, 12, 25).

[64] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A. Sartakov and Peter R. Pietzuch. 2019. SGX-LKL: securing the host OS interface for trusted execution. *CoRR*, abs/1908.11143. http://arxiv.org/abs/1908.11143 arXiv: 1908.11143 (cit. on pp. 3, 6, 12, 25, 82, 83).

[65] Luigi Coppolino, Salvatore D'Antonio, Giovanni Mazzeo and Luigi Romano. 2025. An experimental evaluation of TEE technology: benchmarking transparent approaches based on SGX, SEV, and TDX. *Comput. Secur.*, 154, 104457. DOI: 10.1016/J.COSE.2025.104457 (cit. on pp. 3, 162).

[66] Ayaz Akram, Venkatesh Akella, Sean Peisert and Jason Lowe-Power. 2022. SoK: limitations of confidential computing via TEEs for high-performance compute systems. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED), Storrs, CT, USA, September 26-27, 2022*. IEEE, 121–132. DOI: 10.1109/SEED55351.2022.00018 (cit. on p. 3).

[67] Shengye Wan, Mingshen Sun, Kun Sun, Ning Zhang and Xu He. 2020. RusTEE: developing memory-safe ARM TrustZone applications. In *ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020*. ACM, 442–453. DOI: 10.1145/3427228.3427262 (cit. on pp. 3, 28).

[68] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei and Zhiqiang Lin. 2019. Towards memory safe enclave programming with Rust-SGX. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. ACM, 2333–2350. DOI: 10.1145/3319535.3354241 (cit. on p. 3).

[69] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz and Byoungyoung Lee. 2018. OBLIVIATE: a data oblivious filesystem for Intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. DOI: 10.14722/ndss.2018.23284 (cit. on pp. 3, 91, 109).

[70] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel and Giovanni Vigna. 2017. BOOMERANG: exploiting the semantic gap in trusted execution environments. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. DOI: 10.14722/ndss.2017.23227 (cit. on p. 3).

[71] Maan Haj Rachid, Ryan D. Riley and Qutaibah M. Malluhi. 2020. Enclave-based oblivious RAM using Intel's SGX. *Comput. Secur.*, 91, 101711. DOI: 10.1016/J.COSE.2019.101711 (cit. on p. 3).

[72] Sajin Sasy, Sergey Gorbunov and Christopher W. Fletcher. 2018. ZeroTrace : oblivious memory primitives from Intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. DOI: 10.14722/ndss.2018.23239 (cit. on p. 3).

[73] Pengfei Wu, Qingni Shen, Robert H. Deng, Ximeng Liu, Yinghui Zhang and Zhonghai Wu. 2019. ObliDC: an SGX-based oblivious distributed computing framework with formal proof. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, AsiaCCS 2019, Auckland, New Zealand, July 09-12, 2019*. ACM, 86–99. DOI: 10.1145/3321705.3329822 (cit. on p. 3).

[74] Tuba Yavuz, Farhaan Fowze, Grant Hernandez, Ken Yihang Bai, Kevin R. B. Butler and Dave Jing Tian. 2023. ENCIDER: detecting timing and cache side channels in SGX enclaves and cryptographic APIs. *IEEE Trans. Dependable Secur. Comput.*, 20, 2, 1577–1595. DOI: 10.1109/TDSC.2022.3160346 (cit. on p. 3).

[75] Marcel Busch, Aravind Machiry, Chad Spensky, Giovanni Vigna, Christopher Kruegel and Mathias Payer. 2023. TEEzz: fuzzing trusted applications on COTS Android devices. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 1204–1219. DOI: 10.1109/SP46215.2023.10179302 (cit. on p. 3).

[76] Tobias Cloosters, Johannes Willbold, Thorsten Holz and Lucas Davi. 2022. SGXFuzz: efficiently synthesizing nested structures for SGX enclave fuzzing. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. USENIX Association, 3147–3164. https://www.usenix.org/conference/usenixsecurity22/presentation/cloosters (cit. on p. 3).

[77] Luigi Coppolino, Salvatore D'Antonio, Giovanni Mazzeo and Luigi Romano. 2019. A comparative analysis of emerging approaches for securing Java software with Intel SGX. *Future Gener. Comput. Syst.*, 97, 620–633. DOI: 10.1016/J.FUTURE.2019.03.018 (cit. on p. 4).

[78] Jianyu Jiang, Xusheng Chen, Tsz On Li, Cheng Wang, Tianxiang Shen, Shixiong Zhao, Heming Cui, Cho-Li Wang and Fengwei Zhang. 2020. Uranus: simple, efficient SGX programming and its applications. In *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020*. ACM, 826–840. DOI: 10.1145/3320269.3384763 (cit. on p. 4).

[79] Nuno Santos, Himanshu Raj, Stefan Saroiu and Alec Wolman. 2014. Using Arm TrustZone to build a trusted language runtime for mobile applications. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014*. ACM, 67–80. DOI: 10.1145/2541940.2541949 (cit. on pp. 4, 84).

[80] Chia-che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa and Donald E. Porter. 2020. Civet: an efficient Java partitioning framework for hardware enclaves. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. USENIX Association, 505–522. https://www.usenix.org/conference/usenixsecurity20/presentation/tsai (cit. on pp. 4, 6, 84).

[81] Mingyu Wu, Zhe Li, Haibo Chen, Binyu Zang, Shaojun Wang, Lei Yu, Sanhong Li and Haitao Song. 2024. Toward an SGX-friendly Java runtime. *IEEE Trans. Computers*, 73, 1, 44–57. DOI: 10.1109/TC.2023.3318400 (cit. on p. 4).

[82] K.V. Nori. 1974. The PASCAL "P" compiler. implementation notes. Report. Technical Reports D-INFK. Zürich. DOI: 10.3929/ethz-a-000142255 (cit. on p. 4).

[83] Niklaus Wirth. 1975. PASCAL-S. a subset and its implementation. en. Report. Technical Reports D-INFK. Zürich. DOI: 10.3929/ethz-a-000147073 (cit. on p. 4).

[84] Jack W. Davidson and Christopher W. Fraser. 1984. Register allocation and exhaustive peephole optimization. *Softw. Pract. Exp.*, 14, 9, 857–865. DOI: 10.1002/SPE.4380140906 (cit. on p. 4).

[85] GNU Project. 2001. Contributors of gcc 2.95.3. (17 Mar. 2001). Retrieved 19 Aug. 2024 from https://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc_23.html (cit. on p. 4).

[86] GNU Project. 2024. GCC, the GNU compiler collection. (1 Aug. 2024). Retrieved 19 Aug. 2024 from https://gcc.gnu.org/ (cit. on p. 4).

[87] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley and Daniel Smith. 2024. *The Java Virtual Machine Specification*. (Java SE 22 ed.). Oracle. (9 Feb. 2024). https://docs.oracle.com/javase/specs/jvms/se22/html/index.html (cit. on p. 4).

[88] ISO/IEC. 2012. Information technology – Common Language Infrastructure (CLI). Standard ISO/IEC 23271:2012. International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), (15 Feb. 2012). https://www.iso.org/standard/58046.html (cit. on p. 4).

[89] Microsoft Corporation. 2022. .NET. What is .NET? (1 Nov. 2022). Retrieved 10 June 2024 from https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet (cit. on p. 4).

[90] Microsoft Corporation. 2022. .NET framework. What is .NET Framework? (1 Nov. 2022). Retrieved 10 June 2024 from https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet-framework (cit. on p. 4).

[91] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula and Nicholas Fullagar. 2009. Native Client: A sandbox for portable, untrusted x86 native code. In *30th IEEE Symposium on Security and Privacy (SP 2009), 17-20 May 2009, Oakland, California, USA*. IEEE Computer Society, 79–93. DOI: 10.1109/SP.2009.25 (cit. on p. 4).

[92] Alan Donovan, Robert Muth, Brad Chen and David Sehr. 2010. PNaCl: Portable native client executables. Tech. rep. Retrieved 10 June 2024 from https://css.csail.mit.edu/6.858/2015/readings/pnacl.pdf (cit. on p. 4).

[93] David Herman, Luke Wagner and Alon Zakai. 2014. *asm.js specifications*. Work draft. Mozilla. (18 Aug. 2014). Retrieved 10 June 2024 from http://asmjs.org/spec/latest/ (cit. on p. 4).

[94] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai and J. F. Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. ACM, 185–200. DOI: 10.1145/3062341.3062363 (cit. on pp. 4, 11, 57).

[95] João de Macedo, Rui Abreu, Rui Pereira and João Saraiva. 2022. WebAssembly versus JavaScript: energy and runtime performance. In *International Conference on ICT for Sustainability, ICT4S 2022, Plovdiv, Bulgaria, June 13-17, 2022*. IEEE, 24–34. DOI: 10.1109/ICT4S55073.2022.00014 (cit. on p. 5).

[96] Wenwen Wang. 2022. How far we've come - a characterization study of standalone WebAssembly runtimes. In *IEEE International Symposium on Workload Characterization, IISWC 2022, Austin, TX, USA, November 6-8, 2022*. IEEE, 228–241. DOI: 10.1109/IISWC55918.2022.00028 (cit. on pp. 5, 6).

[97] Mozilla. 2019. Standardizing WASI: a system interface to run WebAssembly outside the web. (27 Mar. 2019). Retrieved 10 June 2024 from https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/ (cit. on pp. 5, 62).

[98] Bytecode Alliance. 2024. Building a minimal Wasmtime embedding. (30 Mar. 2024). Retrieved 11 June 2024 from https://github.com/bytecodealliance/wasmtime/blob/main/docs/examples-minimal.md (cit. on p. 5).

[99] Bytecode Alliance. 2024. Build WAMR vmcore. (14 May 2024). Retrieved 11 June 2024 from https://github.com/bytecodealliance/wasm-micro-runtime/blob/main/doc/build_wamr.md (cit. on p. 5).

[100] Jack B. Dennis and Earl C. Van Horn. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9, 3, 143–155. DOI: 10.1145/365230.365252 (cit. on pp. 5, 63).

[101] Martín Abadi, Mihai Budiu, Úlfar Erlingsson and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13, 1, 4:1–4:40. DOI: 10.1145/1609956.1609960 (cit. on p. 5).

[102] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue and Uday R. Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013*. ACM, 10. DOI: 10.1145/2487726.2488368 (cit. on pp. 5, 22, 23, 82, 112).

[103] Victor Costan, Ilia A. Lebedev and Srinivas Devadas. 2016. Sanctum: minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. USENIX Association, 857–874. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan (cit. on pp. 5, 22, 34, 82, 112, 113, 142).

[104] Tiago Alves and Don Felton. 2004. TrustZone: Integrated Hardware and Software Security. Enabling Trusted Computing in Embedded Systems. Tech. rep. Arm, (15 July 2004). 12 pp. Retrieved 12 June 2024 from https://web.archive.org/web/20050309050300/http://www.arm.com/pdfs/TZ%20Whitepaper.pdf (cit. on pp. 5, 21, 22, 27, 82, 112).

[105] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic and Dawn Song. 2020. Keystone: an open framework for architecting trusted execution environments. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*. ACM, 38:1–38:16. DOI: 10.1145/3342195.3387532 (cit. on pp. 5, 21, 34, 82, 112, 142).

[106] David Kaplan. 2023. Hardware VM isolation in the cloud: enabling confidential computing with AMD SEV-SNP technology. *ACM Queue*, 21, 4, 49–67. DOI: 10.1145/3623392 (cit. on pp. 6, 22, 160).

[107] Intel Corporation. 2020. Intel Trust Domain Extensions. Tech. rep. Version 4. (3 Aug. 2020). 9 pp. Retrieved 29 July 2024 from https://web.archive.org/web/20200817100702/https://software.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf (cit. on pp. 6, 22, 35, 82, 112, 153, 160).

[108] Arm Holdings plc. 2021. Arm CCA Security Model. White paper DEN0096. (2 Aug. 2021). 68 pp. Retrieved 12 June 2024 from https://documentation-service.arm.com/static/610aaec33d73a34b640e333b (cit. on pp. 6, 22, 35, 82, 112, 113, 153, 160).

[109] Shweta Shinde, Dat Le Tien, Shruti Tople and Prateek Saxena. 2017. Panoply: low-TCB Linux applications with SGX enclaves. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/panoply-low-tcb-linux-applications-sgx-enclaves/ (cit. on p. 6).

[110] Subhas Chandra Misra and Virendra C. Bhavsar. 2003. Relationships between selected software measures and latent bug-density: guidelines for improving quality. In *Computational Science and Its Applications - ICCSA 2003, International Conference, Montreal, Canada, May 18-21, 2003, Proceedings, Part I* (Lecture Notes in Computer Science). Vol. 2667. Springer, 724–732. DOI: 10.1007/3-540-44839-X_76 (cit. on pp. 6, 21).

[111] Joakim Bech. 2014. OP-TEE, open-source security for the mass-market. Linaro Limited. (3 Sept. 2014). Retrieved 12 June 2024 from https://web.archive.org/web/20200802203126/https://www.linaro.org/blog/op-tee-open-source-security-mass-market/ (cit. on pp. 6, 27, 30, 112).

[112] Christian Göttel. 2022. *On the challenges of energy efficiency, scalability and security for internet of things services*. PhD thesis. University of Neuchâtel. DOI: 10.35662/unine-thesis-2952 (cit. on pp. 6, 11).

[113] Sébastien Vaucher, Rafael Pires, Pascal Felber, Marcelo Pasin, Valerio Schiavoni and Christof Fetzer. 2018. SGX-aware container orchestration for heterogeneous clusters. In *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*. IEEE Computer Society, 730–741. DOI: 10.1109/ICDCS.2018.00076 (cit. on p. 6).

[114] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Christof Fetzer and Peter R. Pietzuch. 2017. Glamdring: automatic application partitioning for Intel SGX. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. USENIX Association, 285–298. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind (cit. on p. 6).

[115] Peterson Yuhala, Pascal Felber, Hugo Guiroux, Jean-Pierre Lozi, Alain Tchana, Valerio Schiavoni and Gaël Thomas. 2023. SecV: secure code partitioning via multi-language secure values. In *Proceedings of the 24th International Middleware Conference, Middleware 2023, Bologna, Italy, December 11-15, 2023*. ACM, 207–219. DOI: 10.1145/3590140.3629116 (cit. on p. 6).

[116] Abhinav Jangda, Bobby Powers, Emery D. Berger and Arjun Guha. 2019. Not so fast: analyzing the performance of WebAssembly vs. native code. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association, 107–120. https://www.usenix.org/conference/atc19/presentation/jangda (cit. on pp. 6, 96, 127, 151).

[117] [SW] Bytecode Alliance, WAMR, the WebAssembly Micro Runtime 2019. Retrieved 13 June 2024 from URL: https://github.com/bytecodealliance/wasm-micro-runtime (cit. on pp. 6, 66, 113, 117, 149).

[118] [SW] Steven Massey and Volodymyr Shymanskyy, Wasm3, A fast WebAssembly interpreter and the most universal WASM runtime 2020. Retrieved 13 June 2024 from URL: https://github.com/wasm3/wasm3 (cit. on pp. 6, 65).

[119] Jämes Ménétrey. 2022. WAMR, providing an alternate implementation for wasi file system calls using intel protected file system. (28 Sept. 2022). Retrieved 13 June 2024 from https://github.com/bytecodealliance/wasm-micro-runtime/blob/328fd59/core/shared/platform/linux-sgx/sgx_file.c (cit. on p. 6).

[120] David Goltzsche, Manuel Nieke, Thomas Knauth and Rüdiger Kapitza. 2019. AccTEE: a WebAssembly-based two-way sandbox for trusted resource accounting. In *Proceedings of the 20th International Middleware Conference, Middleware 2019, Davis, CA, USA, December 9-13, 2019*. ACM, 123–135. DOI: 10.1145/3361525.3361541 (cit. on pp. 6, 11, 65, 82, 83, 96, 113, 127).

[121] Franz Gregor, Wojciech Ozga, Sébastien Vaucher, Rafael Pires, Do Le Quoc, Sergei Arnautov, André Martin, Valerio Schiavoni, Pascal Felber and Christof Fetzer. 2020. Trust management as a service: enabling trusted execution in the face of byzantine stakeholders. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*. IEEE, 502–514. DOI: 10.1109/DSN48063.2020.00063 (cit. on pp. 7, 8, 142).

[122] Ittai Anati, Shay Gueron, Simon Johnson and Vincent Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy* number 7. Vol. 13. ACM New York, NY, USA. ACM (cit. on pp. 7, 12, 45, 113).

[123] Robert Buhren, Christian Werling and Jean-Pierre Seifert. 2019. Insecure until proven updated: analyzing AMD SEV's remote attestation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. ACM, 1087–1099. DOI: 10.1145/3319535.3354216 (cit. on pp. 7, 30, 50).

[124] George Coker, Joshua D. Guttman, Peter A. Loscocco, Amy L. Herzog, Jonathan K. Millen, Brian O'Hanlon, John D. Ramsdell, Ariel Segall, Justin Sheehy and Brian T. Sniffen. 2011. Principles of remote attestation. *Int. J. Inf. Sec.*, 10, 2, 63–81. DOI: 10.1007/S10207-011-0124-7 (cit. on p. 7).

[125] Muhammad Usama Sardar, Do Le Quoc and Christof Fetzer. 2020. Towards formalization of enhanced privacy ID (EPID)-based remote attestation in Intel SGX. In *23rd Euromicro Conference on Digital System Design, DSD 2020, Kranj, Slovenia, August 26-28, 2020*. IEEE, 604–607. DOI: 10.1109/DSD51259.2020.00099 (cit. on p. 7).

[126] Muhammad Usama Sardar, Thomas Fossati, Simon Frost and Shale Xiong. 2024. Formal specification and verification of architecturally-defined attestation mechanisms in Arm CCA and Intel TDX. *IEEE Access*, 12, 361–381. DOI: 10.1109/ACCESS.2023.3346501 (cit. on p. 7).

[127] Carlton Shepherd, Raja Naeem Akram and Konstantinos Markantonakis. 2017. Establishing mutually trusted channels for remote sensing devices with trusted execution environments. In *Proceedings of the 12th International Conference on Availability, Reliability and Security, Reggio Calabria, Italy, August 29 - September 01, 2017*. ACM, 7:1–7:10. DOI: 10.1145/3098954.3098971 (cit. on pp. 7, 12, 45, 113, 132, 142).

[128] Ahmad B. Usman, Nigel Cole, Mikael Asplund, Felipe Boeira and Christian Vestlund. 2023. Remote attestation assurance arguments for trusted execution environments. In *Sat-CPS@CODASPY 2023: Proceedings of the 2023 ACM Workshop on Secure and Trustworthy Cyber-Physical Systems, Charlotte, NC, USA, 26 April 2023*. ACM, 33–42. DOI: 10.1145/3579988.3585056 (cit. on p. 7).

[129] Muhammad Usama Sardar and Christof Fetzer. 2023. Confidential computing and related technologies: a critical review. *Cybersecur.*, 6, 1, 10. DOI: 10.1186/S42400-023-00144-1 (cit. on p. 7).

[130] Stephan van Schaik, Alex Seto, Thomas Yurek, Adam Batori, Bader AlBassam, Christina Garman, Daniel Genkin, Andrew Miller, Eyal Ronen and Yuval Yarom. 2022. SoK: SGX.Fail: how stuff get exposed. In Retrieved 18 June 2024 from %5Curl%7Bhttps://sgx.fail%7D (cit. on p. 8).

[131] Arto Niemi, Vasile Adrian Bogdan Pop and Jan-Erik Ekberg. 2021. Trusted sockets layer: a TLS 1.3 based trusted channel protocol. In *Secure IT Systems, 26th Nordic Conference, NordSec 2021, Virtual Event, November 29-30, 2021, Proceedings* (Lecture Notes in Computer Science). Vol. 13115. Springer, 175–191. DOI: 10.1007/978-3-030-91625-1_10 (cit. on pp. 8, 142, 144).

[132] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing and Mona Vij. 2018. Integrating remote attestation with transport layer security. *CoRR*, abs/1801.05863. http://arxiv.org/abs/1801.05863 arXiv: 1801.05863 (cit. on pp. 8, 142).

[133] [SW] Open Enclave SDK contributors, Open Enclave version 0.19.7, 19 June 2024. Retrieved 24 July 2024 from URL: https://github.com/openenclave/openenclave (cit. on pp. 8, 25).

[134] [SW] Inclavare Containers contributors, Inclavare Containers: A novel container runtime, aka confidential container, for cloud-native confidential computing and enclave runtime ecosystem 1 July 2021. Retrieved 18 Aug. 2024 from URL: https://github.com/inclavare-containers/inclavare-containers (cit. on p. 8).

[135] Open Enclave SDK contributors. 2019. Open Enclave, the attested TLS sample. (15 June 2019). Retrieved 18 June 2024 from https://github.com/openenclave/openenclave/tree/master/samples/attested_tls (cit. on pp. 8, 53).

[136] Gramine. 2020. RA-TLS minimal example. (11 June 2020). Retrieved 18 June 2024 from https://github.com/gramineproject/gramine/tree/master/CI-Examples/ra-tls-mbedtls (cit. on p. 8).

[137] [SW] Inclavare Containers contributors, librats: Low level attester and verifier drivers for multiple TEEs 2021. Retrieved 18 June 2024 from URL: https://github.com/inclavare-containers/rats-tls (cit. on pp. 8, 53, 92, 94, 149).

[138] Muhammad Usama Sardar, Arto Niemi, Hannes Tschofenig and Thomas Fossati. A rollercoaster ride on the formal analysis of attested TLS. (30 Jan. 2024). Retrieved 18 June 2024 from https://github.com/CCC-Attestation/meetings/blob/main/materials/MuhammadUsamaSardar_Formal_RA-TLS.pdf (cit. on pp. 8, 161).

[139] [SW] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing and Mona Vij, Github: SGX-RA-TLS proof of concept 20 Nov. 2019. Intel Corporation. Retrieved 18 June 2024 from URL: https://github.com/cloud-security-research/sgx-ra-tls (cit. on p. 8).

[140] NorazahAbd Aziz, Nur Izura Udzir and Ramlan Mahmod. 2014. Extending TLS with mutual attestation for platform integrity assurance. *J. Commun.*, 9, 1, 63–72. DOI: 10.12720/JCM.9.1.63-72 (cit. on pp. 8, 142).

[141] Kevin Walsh and John Manferdelli. 2017. Mechanisms for mutual attested microservice communication. In *Companion Proceedings of the 10th International Conference on Utility and Cloud Computing, UCC 2017, Austin, TX, USA, December 5-8, 2017*. ACM, 59–64. DOI: 10.1145/3147234.3148102 (cit. on pp. 8, 142).

[142] Paul Georg Wagner, Pascal Birnstill and Jürgen Beyerer. 2020. Establishing secure communication channels using remote attestation with TPM 2.0. In *Security and Trust Management - 16th International Workshop, STM 2020, Guildford, UK, September 17-18, 2020, Proceedings* (Lecture Notes in Computer Science). Vol. 12386. Springer, 73–89. DOI: 10.1007/978-3-030-59817-4_5 (cit. on pp. 8, 142).

[143] Confidential Computing Consortium. 2021. The attestation special interest group. (3 Nov. 2021). Retrieved 18 June 2024 from https://github.com/CCC-Attestation/ (cit. on p. 8).

[144] Hannes Tschofenig, Yaron Sheffer, Paul Howard, Ionuț Mihalcea, Yogesh Deshpande, Arto Niemi and Thomas Fossati. 2024. Using Attestation in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). Internet-Draft draft-fossati-tls-attestation-06. Work in Progress. Internet Engineering Task Force, (Mar. 2024). 34 pp. https://datatracker.ietf.org/doc/draft-fossati-tls-attestation/06/ (cit. on pp. 8, 53, 161).

[145] Swiss Post. 2019. IncaMail: encrypt confidential e-mails. Retrieved 21 Aug. 2024 from https://www.post.ch/incamail (cit. on pp. 9, 72).

[146] Rafael Pires. 2020. *Distributed systems and trusted execution environments: Trade-offs and challenges*. PhD thesis. University of Neuchâtel. DOI: 10.35662/unine-thesis-2812 (cit. on p. 11).

[147] Sébastien Vaucher. 2023. *Exploring the potential of modern hardware-assisted security and networking technologies*. PhD thesis. University of Neuchâtel. DOI: 10.35662/unine-thesis-2998 (cit. on p. 11).

[148] Peterson Yuhala. 2024. *Enhancing security and performance in trusted execution environments*. PhD thesis. University of Neuchâtel. DOI: 10.35662/unine-thesis-3104 (cit. on p. 11).

[149] Intel Corporation. 2018. Code sample: Intel SGX remote attestation end-to-end example. (4 July 2018). Retrieved 3 Sept. 2024 from https://www.intel.com/content/www/us/en/developer/articles/code-sample/software-guard-extensions-remote-attestation-end-to-end-example.html (cit. on pp. 12, 118, 119, 142).

[150] Henk Birkholz, Dave Thaler, Michael Richardson, Ned Smith and Wei Pan. 2023. Remote attestation procedures (RATS) architecture. RFC 9334. (Jan. 2023). DOI: 10.17487/RFC9334 (cit. on pp. 12, 40).

[151] HiPEAC. 2022. The HiPEAC tech transfer award of 2022. (1 Sept. 2022). Retrieved 8 Oct. 2024 from https://www.hipeac.net/awards/#/tech-transfer/2022/ (cit. on p. 14).

[152] Bytecode Alliance. 2023. Adding Jämes Ménétrey as a recognized contributor. (12 Dec. 2023). Retrieved 8 Oct. 2024 from https://github.com/bytecodealliance/governance/pull/68 (cit. on p. 14).

[153] Arm Holdings plc. 2021. Arm Realm Management Extension (RME) system architecture. Tech. rep. DEN0129. Version A.a. (23 June 2021). 67 pp. Retrieved 29 July 2024 from https://documentation-service.arm.com/static/60d3309b677cf7536a55bae0 (cit. on pp. 22, 35, 153).

[154] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter R. Pietzuch and Rüdiger Kapitza. 2016. SecureKeeper: confidential ZooKeeper using Intel SGX. In *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016*. ACM, 14. http://dl.acm.org/citation.cfm?id=2988350 (cit. on p. 23).

[155] Meysam Taassori, Ali Shafiee and Rajeev Balasubramonian. 2018. VAULT: reducing paging overheads in SGX with efficient integrity verification structures. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*. ACM, 665–678. DOI: 10.1145/3173162.3177155 (cit. on pp. 23, 25).

[156] Scott Raynor. 2022. What does SGX 2.0 sacrifice in security for better performance? (21 Oct. 2022). Retrieved 23 July 2024 from https://github.com/intel/linux-sgx/issues/899 (cit. on p. 23).

[157] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd and Carlos V. Rozas. 2016. Intel software guard extensions support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016, HASP@ICSA 2016, Seoul, Republic of Korea, June 18, 2016*. ACM, 10:1–10:9. DOI: 10.1145/2948618.2954331 (cit. on pp. 23, 89).

[158] Intel Corporation. 2023. Which platforms support Intel software guard extensions SGX2? (12 Dec. 2023). Retrieved 23 July 2024 from https://www.intel.com/content/www/us/en/support/articles/000058764/software/intel-security-products.html (cit. on p. 23).

[159] Intel Corporation. 2024. *Intel 64 and IA-32 Architectures Software Developer's Manual*. 325384-084US. Volume 3. (1 June 2024). 1532 pp. (cit. on p. 23).

[160] Simon Johnson, Raghunandan Makaram, Amy Santoni and Vinnie Scarlata. 2021. Supporting Intel SGX on multi-socket platforms, (24 Mar. 2021). https://web.archive.org/web/20230302235023/https://intel.com/content/dam/www/public/us/en/documents/white-papers/supporting-intel-sgx-on-mulit-socket-platforms.pdf (cit. on pp. 24, 97).

[161] Pierre Louis Aublin, Mohammad Mahhouk and Rüdiger Kapitza. 2022. Towards TEEs with large secure memory and integrity protection against HW attacks. In *SysTEX '22: 5th Workshop on System Software for Trusted Execution, February 28, 2022, Lausanne, Switzerland* (SysTEX '22). Lausanne, Switzerland. https://systex22.github.io/papers/systex22-final15.pdf (cit. on p. 24).

[162] Intel Corporation. 2015. Intel software guard extensions. In *Tutorials of the 42nd International Symposium on Computer Architecture (ISCA'15), Portland, OR, USA, June 13 - 17, 2015* number 332680-002. (25 June 2015). Retrieved 23 July 2024 from https://community.intel.com/legacyfs/online/drupal_files/332680-002.pdf (cit. on p. 24).

[163] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, 86. http://eprint.iacr.org/2016/086 (cit. on pp. 24, 106, 127).

[164] Intel Corporation. 2022. Intel Architecture Memory Encryption Technologies. Tech. rep. 336907-004US. Version 1.4. (1 Aug. 2022). 33 pp. Retrieved 23 July 2024 from https://cdrdv2-public.intel.com/679154/multi-key-total-memory-encryption-spec-1.4.pdf (cit. on pp. 24, 97).

[165] Intel Corporation. 2024. *Intel Software Guard Extensions SDK for Linux OS. Developer Reference*. Version 2.24. (1 Apr. 2024). 484 pp. Retrieved 24 July 2024 from https://download.01.org/intel-sgx/sgx-linux/2.24/docs/Intel_SGX_Developer_Reference_Linux_2.24_Open_Source.pdf (cit. on p. 25).

[166] [SW] Apache Software Foundation, Teaclave SGX SDK version 1.1.3, 25 Oct. 2020. Retrieved 24 July 2024 from URL: https://github.com/apache/incubator-teaclave-sgx-sdk (cit. on p. 25).

[167] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia and Shoumeng Yan. 2020. Occlum: secure and efficient multitasking inside a single enclave of Intel SGX. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*. ACM, 955–970. DOI: 10.1145/3373376.3378469 (cit. on p. 25).

[168] Intel Corporation. 2021. XuCode: an innovative technology for implementing complex instruction flows. (6 May 2021). Retrieved 25 July 2024 from https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/xucode-implementing-complex-instruction-flows.html (cit. on p. 26).

[169] Maxim Goryachy, Dmitry Sklyarov and Mark Ermolov. 2022. Chip red pill: how we achieved the arbitrary [micro]code execution inside Intel Atom CPUs. In *Offensive Security Conference (OffensiveCon'22)* (Berlin, Germany). (5 Feb. 2022). Retrieved 25 July 2024 from https://www.offensivecon.org/speakers/2022/maxim-goryachy.html (cit. on p. 26).

[170] [SW] Maxim Goryachy, Dmitry Sklyarov and Mark Ermolov, MicrocodeDecryptor 19 July 2022. Retrieved 25 July 2024 from URL: https://github.com/chip-red-pill/MicrocodeDecryptor (cit. on p. 26).

[171] [SW] Intel Corporation, linux-sgx version 2.24, 28 Apr. 2024. Retrieved 25 July 2024 from URL: https://github.com/intel/linux-sgx (cit. on p. 26).

[172] Arm Holdings plc. 2024. *Learn the architecture – TrustZone for AArch64*. 102418_0101_03_en. Version 1.1. (23 Feb. 2024). 53 pp. Retrieved 25 July 2024 from https://documentation-service.arm.com/static/65d87a8a837c4d065f654550 (cit. on p. 27).

[173] Arm Holdings plc. 2018. *Arm TrustZone Technology for the Armv8-M Architecture*. 100690_0201_00_en. Version 2.1. (30 Oct. 2018). 26 pp. Retrieved 25 July 2024 from https://documentation-service.arm.com/static/5f873034f86e16515cdb6d3e (cit. on p. 27).

[174] Sandro Pinto, Jorge Pereira, Tiago Gomes, Mongkol Ekpanyapong and Adriano Tavares. 2017. Towards a TrustZone-assisted hypervisor for real-time embedded systems. *IEEE Comput. Archit. Lett.*, 16, 2, 158–161. DOI: 10.1109/LCA.2016.2617308 (cit. on p. 29).

[175] José Martins, João Alves, Jorge Cabral, Adriano Tavares and Sandro Pinto. 2017. µRTZVisor: a secure and safe real-time hypervisor. *Electronics*, 6, 4. DOI: 10.3390/electronics6040093 (cit. on p. 29).

[176] [SW] Arm Holdings plc, Trusted Firmware for A profile Arm CPUs version 2.11.0, 17 May 2024. Retrieved 26 July 2024 from URL: https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git/ (cit. on p. 29).

[177] [SW] Arm Holdings plc, Trusted Firmware for M profile Arm CPUs version 2.1.0, 13 May 2024. Retrieved 26 July 2024 from URL: https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/ (cit. on p. 29).

[178] Brian McGillion, Tanel Dettenborn, Thomas Nyman and N. Asokan. 2015. Open-TEE – an open virtual trusted execution environment. In *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*. IEEE, 400–407. DOI: 10.1109/TRUSTCOM.2015.400 (cit. on p. 30).

[179] Sandro Pinto and Nuno Santos. 2019. Demystifying Arm TrustZone: A comprehensive survey. *ACM Comput. Surv.*, 51, 6, 130:1–130:36. DOI: 10.1145/3291047 (cit. on p. 30).

[180] Xilinx. 2014. *Programming ARM TrustZone Architectureon the Xilinx Zynq-7000 All Programmable SoC. User guide*. UG1019. Version 1.0. (6 May 2014). 44 pp. Retrieved 26 July 2024 from https://docs.amd.com/api/khub/documents/_iwi5NjZs8Xt_NPXeWh_PQ/content (cit. on p. 30).

[181] Johannes Winter. 2012. Experimenting with ARM TrustZone - or: how I met friendly piece of trusted hardware. In *11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2012, Liverpool, United Kingdom, June 25-27, 2012*. IEEE Computer Society, 1161–1166. DOI: 10.1109/TRUSTCOM.2012.157 (cit. on p. 30).

[182] David Kaplan, Jeremy Powell and Tom Woller. 2016. AMD memory encryption. White paper. Version 7. AMD, (21 Apr. 2016). 12 pp. Retrieved 12 June 2024 from https://web.archive.org/web/20160509095844/https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf (cit. on pp. 30, 50, 82, 112).

[183] Felicitas Hetzelt and Robert Buhren. 2017. Security analysis of encrypted virtual machines. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2017, Xi'an, China, April 8-9, 2017*. ACM, 129–142. DOI: 10.1145/3050748.3050763 (cit. on p. 30).

[184] David Kaplan. 2017. Protecting VM register state with SEV-ES. White paper. AMD, (17 Feb. 2017). 8 pp. Retrieved 26 July 2024 from https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/Protecting-VM-Register-State-with-SEV-ES.pdf (cit. on pp. 30, 82, 112).

[185] AMD. 2020. AMD SEV-SNP: strengthening VM isolation with integrity protection and more. White paper. (1 Jan. 2020). 20 pp. Retrieved 26 July 2024 from https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/Protecting-VM-Register-State-with-SEV-ES.pdf (cit. on pp. 30, 44, 50, 82, 112).

[186] Mathias Morbitzer, Sergej Proskurin, Martin Radev, Marko Dorfhuber and Erick Quintanar Salas. 2021. SEVerity: code injection attacks against encrypted virtual machines. In *IEEE Security and Privacy Workshops, SP Workshops 2021, San Francisco, CA, USA, May 27, 2021*. IEEE, 444–455. DOI: 10.1109/SPW53761.2021.00063 (cit. on p. 30).

[187] Luca Wilke, Jan Wichelmann, Florian Sieck and Thomas Eisenbarth. 2021. undeSErVed trust: exploiting permutation-agnostic remote attestation. In *IEEE Security and Privacy Workshops, SP Workshops 2021, San Francisco, CA, USA, May 27, 2021*. IEEE, 456–466. DOI: 10.1109/SPW53761.2021.00064 (cit. on p. 30).

[188] AMD. 2023. *Using SEV with AMD EPYC Processors. User guide*. 58207. Version 1.1. (3 Oct. 2023). 46 pp. Retrieved 28 July 2024 from https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/tuning-guides/58207-using-sev-with-amd-epyc-processors.pdf (cit. on p. 33).

[189] AMD. 2023. AMD SEV-TIO: Trusted I/O for Secure Encrypted Virtualization. Tech. rep. (20 Mar. 2023). 15 pp. Retrieved 28 July 2024 from https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/sev-tio-whitepaper.pdf (cit. on p. 33).

[190] Linux kernel. 2024. AMD memory encryption. Version 6.10. (14 July 2024). Retrieved 28 July 2024 from https://www.kernel.org/doc/html/v6.10/arch/x86/amd-memory-encryption.html (cit. on p. 34).

[191] AMD. 2018. Extending secure encrypted virtualization with SEV-ES. In *KVM Forum 2018* (Edinburgh International Conference Centre, Edinburgh, UK). (18 May 2018). Retrieved 28 July 2024 from https://www.linux-kvm.org/images/9/94/Extending-Secure-Encrypted-Virtualization-with-SEV-ES-Thomas-Lendacky-AMD.pdf (cit. on p. 34).

[192] QEMU. 2024. AMD secure encrypted virtualization (SEV). (30 May 2024). Retrieved 28 July 2024 from https://www.qemu.org/docs/master/system/i386/amd-memory-encryption.html (cit. on p. 34).

[193] [SW] AMD and VirTEE, snpguest: A CLI tool for interacting with SEV-SNP guest environment version 0.5.1, 11 June 2024. Retrieved 28 July 2024 from URL: https://github.com/virtee/snpguest (cit. on p. 34).

[194] [SW] AMD and VirTEE, snphost: Administrative utility for SEV-SNP version 0.4.0, 12 July 2024. Retrieved 28 July 2024 from URL: https://github.com/virtee/snphost (cit. on p. 34).

[195] Andrew Waterman, Yunsup Lee, Rimas Avižienis, David Patterson and Krste Asanović, eds. 2024. *The RISC-V Instruction Set Manual, volume II: Privileged Architecture*. Version 20240528. (28 May 2024). 197 pp. (cit. on pp. 34, 160).

[196] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard and Ahmad-Reza Sadeghi. 2019. TIMBER-V: tag-isolated memory bringing fine-grained enclaves to RISC-V. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/timber-v-tag-isolated-memory-bringing-fine-grained-enclaves-to-risc-v/ (cit. on pp. 34, 82, 142).

[197] Ravi Sahita, Vedvyas Shanbhogue, Andrew Bresticker, Atul Khare, Atish Patra, Samuel Ortiz, Dylan Reid and Rajnesh Kanwal. 2023. CoVE: towards confidential computing on RISC-V platforms. In *Proceedings of the 20th ACM International Conference on Computing Frontiers, CF 2023, Bologna, Italy, May 9-11, 2023*. ACM, 315–321. DOI: 10.1145/3587135.3592168 (cit. on pp. 35, 160).

[198] Andrew Bresticker, Andy Dellow, Atish Patra, Atul Khare, Beeman Strong, Christian Bollis, Dingji Li, Dong Du, Dylan Reid, Eckhard Delfs, Fabrice Marinet, Guerney Hunt, Jiewen Yao, Kailun Qin, Manuel Offenberg, Nicholas Wood, Nick Kossifidis, Osman Koyuncu, Qing Li, Rajnesh Kanwal, Rob Bradford, Samuel Ortiz, Vedvyas Shanbhogue, Wojciech Ozga and Yann Loisel. 2024. Confidential VM Extension (CoVE) for Confidential Computing on RISC-V platforms, RISC-V AP-TEE Task Group. Tech. rep. Version 0.6. RISC-V International, (25 Apr. 2024). 135 pp. Retrieved 17 Sept. 2024 from https://github.com/riscv-non-isa/riscv-ap-tee/blob/0ae67422ef4fa1341680b8b5ed457250ebd7e209/specification/riscv-cove.pdf (cit. on pp. 35, 160).

[199] Guerney D. H. Hunt, Ramachandra Pai, Michael V. Le, Hani Jamjoom, Sukadev Bhattiprolu, Rick Boivie, Laurent Dufour, Brad Frey, Mohit Kapur, Kenneth A. Goldman, Ryan Grimm, Janani Janakirman, John M. Ludden, Paul Mackerras, Cathy May, Elaine R. Palmer, Bharata Bhasker Rao, Lawrence Roy, William A. Starke, Jeff Stuecheli, Enriquillo Valdez and Wendel Voigt. 2021. Confidential computing for OpenPOWER. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*. ACM, 294–310. DOI: 10.1145/3447786.3456243 (cit. on p. 35).

[200] Cesare Garlati and Sandro Pinto. 2020. A clean slate approach to Linux security RISC-V enclaves. In *Proceedings of the Embedded World Conference, Nuremberg, Germany* (cit. on pp. 35, 82).

[201] Pascal Nasahl, Robert Schilling, Mario Werner and Stefan Mangard. 2021. HECTOR-V: a heterogeneous CPU architecture for a secure RISC-V execution environment. In *ASIA CCS '21: ACM Asia Conference on Computer and Communications Security, Virtual Event, Hong Kong, June 7-11, 2021*. ACM, 187–199. DOI: 10.1145/3433210.3453112 (cit. on p. 35).

[202] Samuel Lindemer, Gustav Midéus and Shahid Raza. 2020. Real-time thread isolation and trusted execution on embedded RISC-V. In *First International Workshop on Secure RISC-V Architecture Design Exploration* (cit. on p. 35).

[203] Jo Van Bulck, David F. Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia and Frank Piessens. 2019. A tale of two worlds: assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. ACM, 1741–1758. DOI: 10.1145/3319535.3363206 (cit. on p. 36).

[204] David Cerdeira, Nuno Santos, Pedro Fonseca and Sandro Pinto. 2020. SoK: understanding the prevailing security vulnerabilities in TrustZone-assisted TEE systems. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1416–1432. DOI: 10.1109/SP40000.2020.00061 (cit. on pp. 36, 37).

[205] Alexander Nilsson, Pegah Nikbakht Bideh and Joakim Brorsson. 2020. A survey of published attacks on Intel SGX. *CoRR*, abs/2006.13598. https://arxiv.org/abs/2006.13598 arXiv: 2006.13598 (cit. on p. 37).

[206] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li and Yueqiang Cheng. 2021. CIPHERLEAKS: breaking constant-time cryptography on AMD SEV via the ciphertext side channel. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. USENIX Association, 717–732. https://www.usenix.org/conference/usenixsecurity21/presentation/li-mengyuan (cit. on p. 37).

[207] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu and Yinqian Zhang. 2022. A systematic look at ciphertext side channels on AMD SEV-SNP. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 337–351. DOI: 10.1109/SP46214.2022.9833768 (cit. on p. 37).

[208] Wubing Wang, Mengyuan Li, Yinqian Zhang and Zhiqiang Lin. 2023. PwrLeak: exploiting power reporting interface for side-channel attacks on AMD SEV. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 20th International Conference, DIMVA 2023, Hamburg, Germany, July 12-14, 2023, Proceedings* (Lecture Notes in Computer Science). Vol. 13959. Springer, 46–66. DOI: 10.1007/978-3-031-35504-2_3 (cit. on p. 37).

[209] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler and Michael Schwarz. 2024. CacheWarp: software-based fault injection using selective state reset. In *USENIX Security 2024, Philadelphia, PA, USA*. https://www.usenix.org/conference/usenixsecurity24/presentation/zhang-ruiyi (cit. on p. 37).

[210] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi and Shweta Shinde. 2024. HECKLER: breaking confidential VMs with malicious interrupts. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/schl%5C%C3%5C%BCter (cit. on p. 37).

[211] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi and Shweta Shinde. 2024. WeSee: using malicious #VC interrupts to break AMD SEV-SNP. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 4220–4238. DOI: 10.1109/SP54263.2024.00262 (cit. on p. 37).

[212] Arto Niemi, Sampo Sovio and Jan-Erik Ekberg. 2022. Towards interoperable enclave attestation: learnings from decades of academic work. In *31st Conference of Open Innovations Association, FRUCT 2022, Helsinki, Finland, April 27-29, 2022*. IEEE, 189–200. DOI: 10.23919/FRUCT54823.2022.9770907 (cit. on pp. 40–42, 142).

[213] Muhammad Usama Sardar, Thomas Fossati and Simon Frost. 2023. SoK: attestation in confidential computing. *ResearchGate pre-print*, (Jan. 2023) (cit. on pp. 40–42).

[214] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell and Frank Mckeen. 2016. Intel Software Guard Extensions: EPID Provisioning and Attestation Services. White paper. Intel Corporation, (9 Mar. 2016). 10 pp. Retrieved 2 Aug. 2024 from https://community.intel.com/legacyfs/online/drupal_files/managed/57/0e/ww10-2016-sgx-provisioning-and-attestation-final.pdf (cit. on pp. 44, 45).

[215] Vinnie Scarlata, Simon Johnson, James Beaney and Piotr Zmijewski. 2018. Supporting Third Party Attestation for Intel SGX with Intel Data Center Attestation Primitives. White paper. Intel Corporation, (5 Oct. 2018). 8 pp. Retrieved 2 Aug. 2024 from https://cdrdv2-public.intel.com/671314/intel-sgx-support-for-third-party-attestation.pdf (cit. on pp. 44, 45).

[216] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn and Pradeep K. Khosla. 2005. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*. ACM, 1–16. DOI: 10.1145/1095810.1095812 (cit. on p. 44).

[217] Young-Geun Choi, Jeonil Kang and DaeHun Nyang. 2007. Proactive code verification protocol in wireless sensor network. In *Computational Science and Its Applications - ICCSA 2007, International Conference, Kuala Lumpur, Malaysia, August 26-29, 2007. Proceedings, Part II* (Lecture Notes in Computer Science). Vol. 4706. Springer, 1085–1096. DOI: 10.1007/978-3-540-74477-1_97 (cit. on p. 44).

[218] Rodrigo Vieira Steiner and Emil Lupu. 2019. Towards more practical software-based attestation. *Comput. Networks*, 149, 43–55. DOI: 10.1016/J.COMNET.2018.11.003 (cit. on p. 44).

[219] Wenjuan Xu, Xinwen Zhang, Hongxin Hu, Gail-Joon Ahn and Jean-Pierre Seifert. 2012. Remote attestation with domain-based integrity model and policy analysis. *IEEE Trans. Dependable Secur. Comput.*, 9, 3, 429–442. DOI: 10.1109/TDSC.2011.61 (cit. on p. 44).

[220] Joonho Kong, Farinaz Koushanfar, Praveen Kumar Pendyala, Ahmad-Reza Sadeghi and Christian Wachsmann. 2014. PUFatt: embedded platform attestation based on novel processor-based PUFs. In *The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014*. ACM, 109:1–109:6. DOI: 10.1145/2593069.2593192 (cit. on p. 44).

[221] Wei Feng, Yu Qin, Shijun Zhao and Dengguo Feng. 2018. AAoT: lightweight attestation and authentication of low-resource things in IoT and CPS. *Comput. Networks*, 134, 167–182. DOI: 10.1016/J.COMNET.2018.01.039 (cit. on p. 44).

[222] Furkan Turan and Ingrid Verbauwhede. 2019. Propagating trusted execution through mutual attestation. In *Proceedings of the 4th Workshop on System Software for Trusted Execution, SysTEX@SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. ACM, 2:1–2:6. DOI: 10.1145/3342559.3365334 (cit. on p. 45).

[223] Guoxing Chen and Yinqian Zhang. 2022. MAGE: mutual attestation for a group of enclaves without trusted third parties. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. USENIX Association, 4095–4110. https://www.usenix.org/conference/usenixsecurity22/presentation/chen-guoxing (cit. on pp. 45, 113).

[224] Simon Ott, Benjamin Orthen, Alexander Weidinger, Julian Horsch, Vijayanand Nayani and Jan-Erik Ekberg. 2024. MultiTEE: distributing trusted execution environments. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2024, Singapore, July 1-5, 2024*. ACM. DOI: 10.1145/3634737.3637675 (cit. on p. 45).

[225] Ernest F. Brickell, Jan Camenisch and Liqun Chen. 2004. Direct anonymous attestation. In *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004*. ACM, 132–145. DOI: 10.1145/1030083.1030103 (cit. on p. 46).

[226] Intel Corporation. 2024. Intel Software Guard Extensions attestation service using Intel enhanced privacy identification end-of-life timeline. (22 Apr. 2024). Retrieved 2 Aug. 2024 from https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/resources/sgx-ias-using-epid-eol-timeline.html (cit. on p. 48).

[227] Andreas Rossberg. 2024. WebAssembly Core Specification. W3C Working Draft. W3C, (July 2024). Retrieved 7 Aug. 2024 from https://www.w3.org/TR/2024/WD-wasm-core-2-20240718/ (cit. on p. 57).

[228] Phani Kishore Gadepalli, Gregor Peach, Ludmila Cherkasova, Rob Aitken and Gabriel Parmer. 2019. Challenges and opportunities for efficient serverless computing at the edge. In *38th Symposium on Reliable Distributed Systems, SRDS 2019, Lyon, France, October 1-4, 2019*. IEEE, 261–266. DOI: 10.1109/SRDS47363.2019.00036 (cit. on p. 57).

[229] Adam Hall and Umakishore Ramachandran. 2019. An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation, IoTDI 2019, Montreal, QC, Canada, April 15-18, 2019*. ACM, 225–236. DOI: 10.1145/3302505.3310084 (cit. on p. 57).

[230] Philipp Gackstatter, Pantelis A. Frangoudis and Schahram Dustdar. 2022. Pushing serverless to the edge with WebAssembly runtimes. In *22nd IEEE International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2022, Taormina, Italy, May 16-19, 2022*. IEEE, 140–149. DOI: 10.1109/CCGRID54584.2022.00023 (cit. on p. 57).

[231] Vojdan Kjorveziroski and Sonja Filiposka. 2023. WebAssembly as an enabler for next generation serverless computing. *J. Grid Comput.*, 21, 3, 34. DOI: 10.1007/S10723-023-09669-8 (cit. on p. 57).

[232] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova and Gabriel Parmer. 2020. Sledge: a serverless-first, light-weight Wasm runtime for the edge. In *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020*. ACM, 265–279. DOI: 10.1145/3423211.3425680 (cit. on p. 57).

[233] Borui Li and Wei Dong. 2022. Edge-centric programming for IoT applications with automatic code partitioning. *IEEE Trans. Computers*, 71, 10, 2408–2422. DOI: 10.1109/TC.2021.3129367 (cit. on p. 57).

[234] Elliott Wen and Gerald Weber. 2020. Wasmachine: bring IoT up to speed with a WebAssembly OS. In *2020 IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops 2020, Austin, TX, USA, March 23-27, 2020*. IEEE, 1–4. DOI: 10.1109/PERCOMWORKSHOPS48775.2020.9156135 (cit. on p. 57).

[235] Niko Mäkitalo, Tommi Mikkonen, Cesare Pautasso, Victor Bankowski, Paulius Daubaris, Risto Mikkola and Oleg Beletski. 2021. WebAssembly modules as lightweight containers for liquid IoT applications. In *Web Engineering - 21st International Conference, ICWE 2021, Biarritz, France, May 18-21, 2021, Proceedings* (Lecture Notes in Computer Science). Vol. 12706. Springer, 328–336. DOI: 10.1007/978-3-030-74296-6_25 (cit. on p. 57).

[236] Borui Li, Hongchang Fan, Yi Gao and Wei Dong. 2022. Bringing WebAssembly to resource-constrained IoT devices for seamless device-cloud integration. In *MobiSys '22: The 20th Annual International Conference on Mobile Systems, Applications and Services, Portland, Oregon, 27 June 2022 - 1 July 2022*. ACM, 261–272. DOI: 10.1145/3498361.3538922 (cit. on p. 57).

[237] Jonathan Miller. 2018. C# in the browser with blazor. In *MSDN Magazine* number 9. Vol. 33. Microsoft, (1 Sept. 2018). Retrieved 7 Aug. 2024 from https://learn.microsoft.com/en-us/archive/msdn-magazine/2018/september/web-development-csharp-in-the-browser-with-blazor (cit. on p. 58).

[238] [SW] Mono projet, Mono open source ECMA CLI, C# and .NET implementation. Version 6.12.0.206, 13 Feb. 2024. URL: https://github.com/mono/mono (cit. on p. 58).

[239] [SW] W3C WebAssembly Community Group, The WebAssembly Binary Toolkit version 1.0.36, 31 July 2024. Retrieved 5 Oct. 2024 from URL: https://github.com/WebAssembly/wabt (cit. on p. 58).

[240] Chris Lattner and Vikram S. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. DOI: 10.1109/CGO.2004.1281665 (cit. on pp. 58, 61, 112).

[241] Go Authors. 2024. Introduction to the Go compiler. (25 June 2024). Retrieved 5 Oct. 2024 from https://go.dev/src/cmd/compile/README/ (cit. on p. 58).

[242] [SW] TinyGo Authors, TinyGo, Go compiler for small places version 0.33, 20 Aug. 2024. Retrieved 5 Oct. 2024 from URL: https://github.com/WebAssembly/wabt (cit. on p. 58).

[243] [SW] Alexey Andreev, TeaVM, Compiles Java bytecode to JavaScript, WebAssembly and C version 0.10, 30 Apr. 2024. Retrieved 5 Oct. 2024 from URL: https://github.com/konsoletyper/teavm (cit. on p. 58).

[244] Alon Zakai. 2011. Emscripten: an LLVM-to-JavaScript compiler. In *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. ACM, 301–312. DOI: 10.1145/2048147.2048224 (cit. on pp. 58, 75).

[245] Daniel Lehmann, Johannes Kinder and Michael Pradel. 2020. Everything old is new again: binary security of WebAssembly. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. USENIX Association, 217–234. https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann (cit. on p. 60).

[246] [SW] W3C WebAssembly Community Group, WASI libc implementation for WebAssembly version 22, 12 Apr. 2024. Retrieved 12 Aug. 2024 from URL: https://github.com/WebAssembly/wasi-libc (cit. on p. 62).

[247] Yosh Wuyts. 2024. Changes to Rust's WASI targets. Rust Blog. (9 Apr. 2024). Retrieved 12 Aug. 2024 from https://blog.rust-lang.org/2024/04/09/updates-to-rusts-wasi-targets.html (cit. on p. 62).

[248] W3C WebAssembly Community Group. 2019. *The WASI Preview 0 API*. (18 Sept. 2019). Retrieved 13 Aug. 2024 from https://github.com/WebAssembly/WASI/blob/7e38d7d/phases/unstable/witx/wasi_unstable_preview0.witx (cit. on p. 62).

[249] W3C WebAssembly Community Group. 2019. *The WASI Preview 1 API*. (12 Nov. 2019). Retrieved 13 Aug. 2024 from https://github.com/WebAssembly/WASI/blob/1d1358a/phases/snapshot/witx/wasi_snapshot_preview1.witx (cit. on p. 62).

[250] Bytecode Alliance. 2022. How to use Berkeley/POSIX socket apis in WebAssembly. (25 Mar. 2022). Retrieved 13 Aug. 2024 from https://github.com/bytecodealliance/wasm-micro-runtime/blob/106974d/doc/socket_api.md (cit. on p. 62).

[251] W3C WebAssembly Community Group. 2024. *The WASI Preview 2 API*. (24 Jan. 2024). Retrieved 13 Aug. 2024 from https://github.com/WebAssembly/WASI/blob/256b651/preview2/README.md (cit. on p. 63).

[252] Bytecode Alliance. 2024. WASI 0.2 launched. (25 Jan. 2024). Retrieved 18 Aug. 2024 from https://bytecodealliance.org/articles/WASI-0.2 (cit. on p. 63).

[253] W3C WebAssembly Community Group. 2024. Wasi-nn, neural network proposal for WASI. (25 June 2024). Retrieved 13 Aug. 2024 from https://github.com/WebAssembly/wasi-nn (cit. on pp. 63, 160).

[254] [SW] Google LLC, V8: Google's open source JavaScript/WebAssembly engine version 12.9.66450138, 15 Aug. 2024. Retrieved 15 Aug. 2024 from URL: https://github.com/v8/v8 (cit. on p. 65).

[255] [SW] Mozilla, SpiderMonkey: Mozilla's JavaScript and WebAssembly Engine 16 July 2024. Retrieved 15 Aug. 2024 from URL: https://spidermonkey.dev/ (cit. on p. 65).

[256] [SW] Andrew Scheidecker, WAVM, A WebAssembly Virtual Machine 14 May 2022. Retrieved 16 Aug. 2024 from URL: https://github.com/WAVM/WAVM (cit. on p. 65).

[257] [SW] Wasmer Incorporation, Wasmer, the leading WebAssembly Runtime supporting WASIX, WASI and Emscripten 16 July 2024. Retrieved 16 Aug. 2024 from URL: https://github.com/wasmerio/wasmer (cit. on p. 66).

[258] [SW] Bytecode Alliance, Cranelift, a fast, secure, relatively simple and innovative compiler backend 12 Aug. 2024. Retrieved 16 Aug. 2024 from URL: https://github.com/bytecodealliance/wasmtime/tree/main/cranelift (cit. on p. 66).

[259] [SW] Wasmer Incorporation, Singlepass, compiler implementation based on the Singlepass linear compiler 16 July 2024. Retrieved 16 Aug. 2024 from URL: https://github.com/wasmerio/wasmer/tree/491cba4/lib/compiler-singlepass (cit. on p. 66).

[260] [SW] WasmEdge Community, WasmEdge, a lightweight, high-performance, and extensible WebAssembly runtime for cloud native, edge, and decentralized applications 23 May 2024. Retrieved 16 Aug. 2024 from URL: https://github.com/WasmEdge/WasmEdge (cit. on p. 66).

[261] [SW] Bytecode Alliance, Singlepass, a fast and secure runtime for WebAssembly 12 Aug. 2024. Retrieved 16 Aug. 2024 from URL: https://github.com/bytecodealliance/wasmtime (cit. on p. 66).

[262] [SW] Bytecode Alliance, Lucet, the Sandboxing WebAssembly Compiler 18 Feb. 2020. Retrieved 16 Aug. 2024 from URL: https://github.com/bytecodealliance/lucet (cit. on p. 66).

[263] Damian Poddebniak, Christian Dresen, Jens Müller, Fabian Ising, Sebastian Schinzel, Simon Friedberger, Juraj Somorovsky and Jörg Schwenk. 2018. Efail: breaking S/MIME and OpenPGP email encryption using exfiltration channels. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. USENIX Association, 549–566. https://www.usenix.org/conference/usenixsecurity18/presentation/poddebniak (cit. on pp. 72, 73).

[264] Jonathan Katz and Bruce Schneier. 2000. A chosen ciphertext attack against several e-mail encryption protocols. In *9th USENIX Security Symposium, Denver, Colorado, USA, August 14-17, 2000*. USENIX Association. https://www.usenix.org/conference/9th-usenix-security-symposium/chosen-ciphertext-attack-against-several-e-mail-encryption (cit. on p. 73).

[265] Jens Müller, Marcus Brinkmann, Damian Poddebniak, Sebastian Schinzel and Jörg Schwenk. 2019. Re: what's up Johnny? - covert content attacks on email end-to-end encryption. In *Applied Cryptography and Network Security - 17th International Conference, ACNS 2019, Bogota, Colombia, June 5-7, 2019, Proceedings* (Lecture Notes in Computer Science). Vol. 11464. Springer, 24–42. DOI: 10.1007/978-3-030-21568-2_2 (cit. on p. 73).

[266] Jörg Schwenk, Marcus Brinkmann, Damian Poddebniak, Jens Müller, Juraj Somorovsky and Sebastian Schinzel. 2020. Mitigation of attacks on email end-to-end encryption. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. ACM, 1647–1664. DOI: 10.1145/3372297.3417878 (cit. on p. 73).

[267] Microsoft Corporation. 2024. Create a Blazor WebAssembly Outlook add-in. (10 July 2024). Retrieved 20 Aug. 2024 from https://github.com/OfficeDev/Office-Add-in-samples/tree/a7e4a94/Samples/blazor-add-in/outlook-blazor-add-in (cit. on p. 73).

[268] Harry Halpin. 2014. The W3C web cryptography API: motivation and overview. In *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014, Companion Volume*. ACM, 959–964. DOI: 10.1145/2567948.2579224 (cit. on p. 74).

[269] [SW] OpenSSL Project community, OpenSSL, TLS/SSL and cryptographic library version 3.3.1, 4 June 2024. Retrieved 21 Aug. 2024 from URL: https://github.com/openssl/openssl (cit. on p. 74).

[270] W3C WebAssembly Community Group. 2024. Wasi-crypto, WASI Cryptography API proposal. (2 May 2024). Retrieved 20 Aug. 2024 from https://github.com/WebAssembly/wasi-crypto (cit. on pp. 75, 90, 160).

[271] LambdaTest. 2017. Cross browser testing cloud. Retrieved 21 Aug. 2024 from https://www.lambdatest.com (cit. on p. 77).

[272] [SW] Apple Inc., WebKit, the browser engine used by Safari, Mail, App Store and many other applications on macOS, iOS and Linux version 2.45.90, 19 Aug. 2024. Retrieved 21 Aug. 2024 from URL: https://github.com/WebKit/WebKit (cit. on p. 77).

[273] Chromium project. 2023. Blink, rendering engine. Retrieved 21 Aug. 2024 from https://www.chromium.org/blink/ (cit. on p. 77).

[274] Chromium blog. 2017. Open-sourcing Chrome on iOS. (31 Jan. 2017). Retrieved 21 Aug. 2024 from https://blog.chromium.org/2017/01/open-sourcing-chrome-on-ios.html (cit. on p. 77).

[275] Emily Stark, Michael Hamburg and Dan Boneh. 2009. Symmetric cryptography in javascript. In *Twenty-Fifth Annual Computer Security Applications Conference, ACSAC 2009, Honolulu, Hawaii, USA, 7-11 December 2009*. IEEE Computer Society, 373–381. DOI: 10.1109/ACSAC.2009.42 (cit. on p. 77).

[276] Gang Tan. 2017. Principles and implementation techniques of software-based fault isolation. *Found. Trends Priv. Secur.*, 1, 3, 137–198. DOI: 10.1561/3300000013 (cit. on p. 82).

[277] Jay Bosamiya, Wen Shih Lim and Bryan Parno. 2022. Provably-safe multilingual software sandboxing using WebAssembly. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. USENIX Association, 1975–1992. https://www.usenix.org/conference/usenixsecurity22/presentation/bosamiya (cit. on p. 82).

[278] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter and Emmett Witchel. 2017. Ryoan: a distributed sandbox for untrusted computation on secret data. *ACM Trans. Comput. Syst.*, 35, 4, 13:1–13:32. DOI: 10.1145/3231594 (cit. on pp. 82, 84).

[279] Weijie Liu, Wenhao Wang, Hongbo Chen, Xiaofeng Wang, Yaosong Lu, Kai Chen, Xinyu Wang, Qintao Shen, Yi Chen and Haixu Tang. 2021. Practical and efficient in-enclave verification of privacy compliance. In *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021, Taipei, Taiwan, June 21-24, 2021*. IEEE, 413–425. DOI: 10.1109/DSN48987.2021.00052 (cit. on pp. 82, 84).

[280] Intel Corporation. 2016. Overview of Intel protected file system library using SGX. (20 Dec. 2016). Retrieved 29 Aug. 2024 from https://intel.ly/34NpzMn (cit. on pp. 82, 91).

[281] Weizhong Qiang, Zezhao Dong and Hai Jin. 2018. Se-Lambda: securing privacy-sensitive serverless applications using SGX enclave. In *Security and Privacy in Communication Networks - 14th International Conference, SecureComm 2018, Singapore, August 8-10, 2018, Proceedings, Part I* (Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering). Vol. 254. Springer, 451–470. DOI: 10.1007/978-3-030-01701-9_25 (cit. on pp. 84, 113).

[282] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. 2016. Serverless computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2016, Denver, CO, USA, June 20-21, 2016*. USENIX Association. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson (cit. on p. 84).

[283] [SW] Enarx Project, Enarx, Confidential Computing with WebAssembly 19 Jan. 2023. Retrieved 29 Aug. 2024 from URL: https://github.com/enarx/enarx (cit. on pp. 84, 113).

[284] Mathias Brossard, Guilhem Bryant, Basma El Gaabouri, Xinxin Fan, Alexandre Ferreira, Edmund Grimley-Evans, Christopher Haster, Evan Johnson, Derek Miller, Fan Mo, Dominic P. Mulligan, Nick Spinale, Eric Van Hensbergen, Hugo J. M. Vincent and Shale Xiong. 2024. Private delegated computations using strong isolation. *IEEE Trans. Emerg. Top. Comput.*, 12, 1, 386–398. DOI: 10.1109/TETC.2023.3281738 (cit. on pp. 84, 113).

[285] Amazon. 2019. AWS Nitro Enclaves. (3 Dec. 2019). Retrieved 29 Aug. 2024 from https://go.aws/3lSjRLx (cit. on pp. 84, 113).

[286] [SW] MesaPy Project Contributors, MesaPy, A Fast and Safe Python based on PyPy 18 Aug. 2018. Retrieved 29 Aug. 2024 from URL: https://github.com/mesalock-linux/mesapy (cit. on p. 84).

[287] David Goltzsche, Colin Wulf, Divya Muthukumaran, Konrad Rieck, Peter R. Pietzuch and Rüdiger Kapitza. 2017. TrustJS: trusted client-side execution of JavaScript. In *Proceedings of the 10th European Workshop on Systems Security, EUROSEC 2017, Belgrade, Serbia, April 23, 2017*. ACM, 7:1–7:6. DOI: 10.1145/3065913.3065917 (cit. on p. 84).

[288] Huibo Wang, Erick Bauman, Vishal Karande, Zhiqiang Lin, Yueqiang Cheng and Yinqian Zhang. 2019. Running language interpreters inside SGX: A lightweight, legacy-compatible script code hardening approach. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, AsiaCCS 2019, Auckland, New Zealand, July 09-12, 2019*. ACM, 114–121. DOI: 10.1145/3321705.3329848 (cit. on p. 84).

[289] Rafael Pires, Daniel Gavril, Pascal Felber, Emanuel Onica and Marcelo Pasin. 2017. A lightweight mapreduce framework for secure processing with SGX. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, Madrid, Spain, May 14-17, 2017*. IEEE Computer Society / ACM, 1100–1107. DOI: 10.1109/CCGRID.2017.129 (cit. on p. 84).

[290] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen and Ahmad-Reza Sadeghi. 2019. DR.SGX: automated and adjustable side-channel protection for SGX using data location randomization. In *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019*. ACM, 788–800. DOI: 10.1145/3359789.3359809 (cit. on p. 85).

[291] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein and Christof Fetzer. 2018. Varys: protecting SGX enclaves from practical side-channel attacks. In *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. USENIX Association, 227–240. https://www.usenix.org/conference/atc18/presentation/oleksenko (cit. on p. 85).

[292] Ivan Puddu, Moritz Schneider, Daniele Lain, Stefano Boschetto and Srdjan Capkun. 2024. On (the lack of) code confidentiality in trusted execution environments. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 4125–4142. DOI: 10.1109/SP54263.2024.00259 (cit. on pp. 86, 144).

[293] Yanlin Li, Jonathan M. McCune, James Newsome, Adrian Perrig, Brandon Baker and Will Drewry. 2014. MiniBox: a two-way sandbox for x86 native code. In *Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC 2014, Philadelphia, PA, USA, June 19-20, 2014*. USENIX Association, 409–420. https://www.usenix.org/conference/atc14/technical-sessions/presentation/li%5C_yanlin (cit. on p. 87).

[294] Intel Corporation. 2023. *Intel Software Guard Extensions (Intel SGX) SDK for Linux OS — Developer Reference*. version 2.19. (Aug. 2023) (cit. on p. 88).

[295] W3C WebAssembly Community Group. 2020. WebAssembly system interface, WASI application ABI. (7 Oct. 2020). Retrieved 27 Aug. 2024 from https://github.com/WebAssembly/WASI/blob/63f92d1/design/application-abi.md (cit. on p. 89).

[296] [SW] WolfSSL Incorporation, WolfSSL, a small, fast, portable implementation of TLS/SSL for embedded devices to the cloud version 5.7.2, 8 July 2024. Retrieved 29 Aug. 2024 from URL: https://github.com/wolfSSL/wolfssl (cit. on p. 90).

[297] [SW] Cesanta Software Limited, Mongoose, Embedded Web Server version 7.15, 21 Aug. 2024. Retrieved 29 Aug. 2024 from URL: https://github.com/cesanta/mongoose (cit. on p. 90).

[298] Tate Tian. 2017-01-15. Understanding SGX protected file system. Retrieved 6 Apr. 2018 from https://web.archive.org/web/20180406054257/http://www.tatetian.io/2017/01/15/understanding-sgx-protected-file-system/ (cit. on pp. 91, 109).

[299] Jämes Ménétrey. 2022. WAMR, remote attestation and secure channel of communication. (31 Oct. 2022). Retrieved 21 Nov. 2022 from https://github.com/bytecodealliance/wasm-micro-runtime/discussions/1664 (cit. on p. 92).

[300] Zeuson0. 2022. Linux-sgx: improve the remote attestation. (10 Nov. 2022). Retrieved 27 May 2024 from https://github.com/bytecodealliance/wasm-micro-runtime/pull/1695 (cit. on p. 92).

[301] [SW] Louis-Noël Pouchet, PolyBench/C, the Polyhedral Benchmark suite version 4.2.1, 20 May 2016. Retrieved 29 Aug. 2024 from URL: https://sourceforge.net/projects/polybench/ (cit. on pp. 96, 127).

[302] Lv Junyan, Xu Shiguo and Li Yijie. 2009. Application research of embedded database SQLite. In *2009 International Forum on Information Technology and Applications*. Vol. 2, 539–543. DOI: 10.1109/IFITA.2009.408 (cit. on pp. 100, 128).

[303] D. Richard Hipp and Ginger G. Wyrick. 2022. SQLite, measuring and reducing CPU usage. (8 Jan. 2022). Retrieved 29 Aug. 2024 from https://sqlite.org/cpu.html (cit. on pp. 101, 128).

[304] Vasily A. Sartakov, Nico Weichbrodt, Sebastian Krieter, Thomas Leich and Rüdiger Kapitza. 2018. STANlite - a database engine for secure data processing at rack-scale level. In *2018 IEEE International Conference on Cloud Engineering, IC2E 2018, Orlando, FL, USA, April 17-20, 2018*. IEEE Computer Society, 23–33. DOI: 10.1109/IC2E.2018.00024 (cit. on p. 102).

[305] Intel Corporation. 2019. Performance Considerations for Intel Software Guard Extensions Applications. White paper. (23 Apr. 2019). Retrieved 29 Aug. 2024 from https://www.intel.com/content/www/us/en/content-details/671502/performance-considerations-for-intel-software-guard-extensions-intel-sgx-applications.html (cit. on p. 102).

[306] [SW] Yuji Hirose, Cpp-httplib, a C++ header-only HTTP/HTTPS server and client library version 0.16.3, 17 Aug. 2024. Retrieved 29 Aug. 2024 from URL: https://github.com/yhirose/cpp-httplib (cit. on p. 107).

[307] WolfSSL Incorporation. 2024. WolfSSL for Intel SGX. (8 Mar. 2024). Retrieved 29 Aug. 2024 from https://github.com/wolfSSL/wolfssl/tree/master/IDE/LINUX-SGX (cit. on p. 107).

[308] [SW] Niels Lohmann, Nlohmann, JSON for Modern C++ version 3.11.3, 28 Nov. 2023. Retrieved 29 Aug. 2024 from URL: https://github.com/nlohmann/json (cit. on p. 108).

[309] Intel Corporation. 2016. Introduction to Intel SGX sealing. (4 May 2016). Retrieved 29 Aug. 2024 from https://intel.com/content/www/us/en/developer/articles/technical/introduction-to-intel-sgx-sealing.html (cit. on p. 109).

[310] IoT Analytics. 2020. State of the IoT 2020: 12 billion IoT connections. (19 Nov. 2020). Retrieved 3 Sept. 2024 from https://iot-analytics.com/state-of-the-iot-2020-12-billion-iot-connections-surpassing-non-iot-for-the-first-time/ (cit. on p. 112).

[311] Apple. 2020. Apple unleashes M1. (10 Nov. 2020). Retrieved 3 Sept. 2024 from https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/ (cit. on p. 112).

[312] Kaizheng Liu, Ming Yang, Zhen Ling, Huaiyu Yan, Yue Zhang, Xinwen Fu and Wei Zhao. 2021. On manually reverse engineering communication protocols of Linux-based IoT systems. *IEEE Internet Things J.*, 8, 8, 6815–6827. DOI: 10.1109/JIOT.2020.3036232 (cit. on p. 112).

[313] Zhen Ling, Junzhou Luo, Yiling Xu, Chao Gao, Kui Wu and Xinwen Fu. 2017. Security vulnerabilities of internet of things: a case study of the smart plug system. *IEEE Internet Things J.*, 4, 6, 1899–1909. DOI: 10.1109/JIOT.2017.2707465 (cit. on p. 112).

[314] He Li, Kaoru Ota and Mianxiong Dong. 2018. Learning IoT in edge: deep learning for the internet of things with edge computing. *IEEE Netw.*, 32, 1, 96–101. DOI: 10.1109/MNET.2018.1700202 (cit. on p. 112).

[315] Samsung. 2020. TEEGRIS, a powerful solution to run applications in a trusted execution environment. (Aug. 2020). Retrieved 3 Sept. 2024 from https://developer.samsung.com/teegris/overview.html (cit. on p. 112).

[316] Derek D. Miller. 2022. Status of Intel SGX and Arm TrustZone support. (11 Jan. 2022). https://github.com/veracruz-project/veracruz/issues/330 (cit. on p. 113).

[317] Ferdinand Brasser, Kasper Bonne Rasmussen, Ahmad-Reza Sadeghi and Gene Tsudik. 2016. Remote attestation for low-end embedded devices: the prover's perspective. In *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*. ACM, 91:1–91:6. DOI: 10.1145/2897937.2898083 (cit. on p. 113).

[318] Shijun Zhao, Qianying Zhang, Yu Qin, Wei Feng and Dengguo Feng. 2019. SecTEE: a software-based approach to secure enclave architecture using TEE. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. ACM, 1723–1740. DOI: 10.1145/3319535.3363205 (cit. on p. 113).

[319] Jaehwan Ahn, Il-Gu Lee and Myungchul Kim. 2020. Design and implementation of hardware-based remote attestation for a secure internet of things. *Wirel. Pers. Commun.*, 114, 1, 295–327. DOI: 10.1007/S11277-020-07364-5 (cit. on p. 113).

[320] Zhen Ling, Huaiyu Yan, Xinhui Shao, Junzhou Luo, Yiling Xu, Bryan Pearson and Xinwen Fu. 2021. Secure boot, trusted boot and remote attestation for ARM TrustZone-based IoT nodes. *J. Syst. Archit.*, 119, 102240. DOI: 10.1016/J.SYSARC.2021.102240 (cit. on p. 113).

[321] André Martin, Cong Lian, Franz Gregor, Robert Krahn, Valerio Schiavoni, Pascal Felber and Christof Fetzer. 2021. ADAM-CS: advanced asynchronous monotonic counter service. In *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021, Taipei, Taiwan, June 21-24, 2021*. IEEE, 426–437. DOI: 10.1109/DSN48987.2021.00053 (cit. on p. 115).

[322] Keegan Ryan. 2019. Hardware-backed heist: extracting ECDSA keys from Qualcomm's TrustZone. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. ACM, 181–194. DOI: 10.1145/3319535.3354197 (cit. on p. 115).

[323] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu and Gang Qu. 2019. VoltJockey: breaching TrustZone by software-controlled voltage manipulation over multi-core frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. ACM, 195–209. DOI: 10.1145/3319535.3354201 (cit. on p. 115).

[324] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum and Edward W. Felten. 2008. Lest we remember: cold boot attacks on encryption keys. In *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*. USENIX Association, 45–60. http://www.usenix.org/events/sec08/tech/full%5C_papers/halderman/halderman.pdf (cit. on p. 115).

[325] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou and Y. Thomas Hou. 2018. TruSense: information leakage from TrustZone. In *2018 IEEE Conference on Computer Communications, INFOCOM 2018, Honolulu, HI, USA, April 16-19, 2018*. IEEE, 1097–1105. DOI: 10.1109/INFOCOM.2018.8486293 (cit. on p. 115).

[326] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz and Yuval Yarom. 2019. Spectre attacks: exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 1–19. DOI: 10.1109/SP.2019.00002 (cit. on p. 115).

[327] Ning Zhang, Kun Sun, Wenjing Lou and Yiwei Thomas Hou. 2016. CaSE: cache-assisted secure execution on ARM processors. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 72–90. DOI: 10.1109/SP.2016.13 (cit. on p. 115).

[328] Haehyun Cho, Jinbum Park, Donguk Kim, Ziming Zhao, Yan Shoshitaishvili, Adam Doupé and Gail-Joon Ahn. 2020. SmokeBomb: effective mitigation against cache side-channel attacks on the ARM architecture. In *MobiSys '20: The 18th Annual International Conference on Mobile Systems, Applications, and Services, Toronto, Ontario, Canada, June 15-19, 2020*. ACM, 107–120. DOI: 10.1145/3386901.3388888 (cit. on p. 115).

[329] Jens Wiklander. 2021. OP-TEE, can heap memory contain executable code? (16 Feb. 2021). Retrieved 18 Feb. 2021 from https://github.com/OP-TEE/optee_os/issues/4396 (cit. on p. 117).

[330] NXP Semiconductors N.V. 2018. *Security Reference Manual for i.MX 8M Dual/8M QuadLite/8M Quad*. IMX8MDQLQSRM. Version 0. (Nov. 2018) (cit. on pp. 117, 123).

[331] GlobalPlatform. 2021. *TEE Sockets API Specification*. Version 1.0.2. (Feb. 2021). Retrieved 5 Mar. 2021 from https://globalplatform.org/specs-library/tee-sockets-api-specification/ (cit. on p. 118).

[332] Linaro Limited. 2022. OP-TEE, frequently asked questions. (25 Jan. 2022). Retrieved 25 Jan. 2022 from https://optee.readthedocs.io/en/latest/faq/faq.html (cit. on pp. 118, 121).

[333] Hugo Krawczyk. 2003. SIGMA: the 'SIGn-and-MAc' approach to authenticated Diffie-Hellman and its use in the IKE-protocols. In *Advances in Cryptology, CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings* (Lecture Notes in Computer Science). Vol. 2729. Springer, 400–425. DOI: 10.1007/978-3-540-45146-4_24 (cit. on pp. 118, 142).

[334] Jun Li, Xinman Luo, Hong Lei and Jieren Cheng. 2024. TEEm: supporting large memory for trusted applications in ARM TrustZone. *IEEE Access*, 12, 108584–108596. DOI: 10.1109/ACCESS.2024.3431231 (cit. on p. 121).

[335] Rouven Czerwinski. 2020. OP-TEE is ready: let's use it! In *Embedded Linux Conference 2020*. (27 Oct. 2020). Retrieved 3 Sept. 2024 from https://osseu2020.sched.com/event/eCGo/op-tee-is-ready-lets-use-it-rouven-czerwinski-pengutronix-ek (cit. on p. 123).

[336] Rouven Czerwinski. 2019. OP-TEE: disable CAAM for all i.MX6/7 flavors. (28 Oct. 2019). https://github.com/OP-TEE/optee_os/pull/3349 (cit. on p. 123).

[337] [SW] LibTom Projects Contributors, LibTomCrypt, a fairly comprehensive, modular and portable cryptographic toolkit version 1.18.2, 2 July 2018. Retrieved 3 Sept. 2024 from URL: https://github.com/libtom/libtomcrypt (cit. on p. 123).

[338] Kristin E. Lauter. 2004. The advantages of elliptic curve cryptography for wireless security. *IEEE Wirel. Commun.*, 11, 1, 62–67. DOI: 10.1109/MWC.2004.1269719 (cit. on p. 123).

[339] Elaine Barker. 2020. *Recommendation for key management: part 1 – general*. Number 800-57. (May 2020). DOI: 10.6028/nist.sp.800-57pt1r5 (cit. on p. 123).

[340] [SW] W3C WebAssembly Community Group, WASI-SDK, WASI-enabled WebAssembly C/C++ toolchain version 24, 2 Aug. 2024. Retrieved 3 Sept. 2024 from URL: https://github.com/WebAssembly/wasi-sdk (cit. on pp. 124, 149).

[341] Julien Amacher and Valerio Schiavoni. 2019. On the performance of ARM TrustZone - (practical experience report). In *Distributed Applications and Interoperable Systems - 19th IFIP WG 6.1 International Conference, DAIS 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings* (Lecture Notes in Computer Science). Vol. 11534. Springer, 133–151. DOI: 10.1007/978-3-030-22496-7_9 (cit. on p. 125).

[342] Markus Schordan, Pei-Hung Lin, Daniel J. Quinlan and Louis-Noël Pouchet. 2014. Verification of polyhedral optimizations with constant loop bounds in finite state space computations. In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II* (Lecture Notes in Computer Science). Vol. 8803. Springer, 493–508. DOI: 10.1007/978-3-662-45231-8_41 (cit. on p. 127).

[343] Simon Shillaker and Peter R. Pietzuch. 2020. Faasm: lightweight isolation for efficient stateful serverless computing. In *Proceedings of the 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*. USENIX Association, 419–433. https://www.usenix.org/conference/atc20/presentation/shillaker (cit. on p. 127).

[344] Christof Paar and Jan Pelzl. 2010. *Understanding Cryptography - A Textbook for Students and Practitioners*. Springer. ISBN: 978-3-642-04100-6. DOI: 10.1007/978-3-642-04101-3 (cit. on p. 130).

[345] [SW] Lewis Van Winkle, Genann, simple neural network library in ANSI C version 1.0, 22 Sept. 2020. Retrieved 3 Sept. 2024 from URL: https://github.com/codeplea/genann (cit. on p. 131).

[346] Riccardo Cantoro, Nikolaos Ioannis Deligiannis, Matteo Sonza Reorda, Marcello Traiola and Emanuele Valea. 2020. Evaluating data encryption effects on the resilience of an artificial neural network. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, DFT 2020, Frascati, Italy, October 19-21, 2020*. IEEE, 1–4. DOI: 10.1109/DFT50435.2020.9250869 (cit. on p. 131).

[347] Burin Amornpaisannon, Andreas Diavastos, Li-Shiuan Peh and Trevor E. Carlson. 2020. Laser attack benchmark suite. In *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2020, San Diego, CA, USA, November 2-5, 2020*. IEEE, 50:1–50:9. DOI: 10.1145/3400302.3415646 (cit. on p. 131).

[348] Antony Unwin and Kim Kleinman. 2021. The Iris Data Set: In Search of the Source of Virginica. *Significance*, 18, 6, (Nov. 2021), 26–29. DOI: 10.1111/1740-9713.01589 (cit. on p. 131).

[349] R. A. Fisher. 1988. Iris. UCI Machine Learning Repository. (1988). DOI: 10.24432/C56C76 (cit. on p. 131).

[350] Linaro Limited. 2021. OP-TEE trusted applications: signing of TAs. (19 July 2021). Retrieved 3 Sept. 2024 from https://optee.readthedocs.io/en/latest/building/trusted_applications.html#signing-of-tas (cit. on p. 133).

[351] Etienne Carriere and Jérôme Forissier. 2021. OP-TEE: can an impersonate TA access the storage of another TA by reusing its UUID? (5 Mar. 2021). https://github.com/OP-TEE/optee_os/issues/4447 (cit. on p. 133).

[352] Cas J. F. Cremers. 2008. The Scyther tool: verification, falsification, and analysis of security protocols. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings* (Lecture Notes in Computer Science). Vol. 5123. Springer, 414–418. DOI: 10.1007/978-3-540-70545-1_38 (cit. on p. 133).

[353] Danny Dolev and Andrew Chi-Chih Yao. 1983. On the security of public key protocols. *IEEE Trans. Inf. Theory*, 29, 2, 198–207. DOI: 10.1109/TIT.1983.1056650 (cit. on p. 134).

[354] Cas Cremers and Sjouke Mauw. 2012. *Operational Semantics and Verification of Security Protocols*. *Information Security and Cryptography*. Springer. ISBN: 978-3-540-78636-8. DOI: 10.1007/978-3-540-78636-8 (cit. on p. 134).

[355] G. Lowe. 1997. A hierarchy of authentication specifications. In *Proceedings 10th Computer Security Foundations Workshop*, 31–43. DOI: 10.1109/CSFW.1997.596782 (cit. on p. 134).

[356] Patrick Th. Eugster, Pascal Felber, Rachid Guerraoui and Anne-Marie Kermarrec. 2003. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35, 2, 114–131. DOI: 10.1145/857076.857078 (cit. on p. 138).

[357] Antero Taivalsaari, Tommi Mikkonen and Cesare Pautasso. 2021. Towards seamless IoT device-edge-cloud continuum: software architecture options of IoT devices revisited. In *ICWE 2021 Workshops - ICWE 2021 International Workshops, BECS and Invited Papers, Biarritz, France, May 18-21, 2021, Revised Selected Papers* (Communications in Computer and Information Science). Vol. 1508. Springer, 82–98. DOI: 10.1007/978-3-030-92231-3_8 (cit. on p. 138).

[358] Amazon. 2023. What is pub/sub messaging? (21 Feb. 2023). Retrieved 4 Sept. 2024 from https://aws.amazon.com/pub-sub-messaging (cit. on p. 138).

[359] Google LLC. 2014. Cloud Pub/Sub. (8 Dec. 2014). Retrieved 4 Sept. 2024 from https://cloud.google.com/pubsub (cit. on pp. 138, 139).

[360] Microsoft Corporation. 2022. Publisher-subscriber pattern. (2 Mar. 2022). Retrieved 4 Sept. 2024 from https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber (cit. on p. 138).

[361] Emanuel Onica, Pascal Felber, Hugues Mercier and Etienne Rivière. 2016. Confidentiality-preserving publish/subscribe: a survey. *ACM Comput. Surv.*, 49, 2, 27:1–27:43. DOI: 10.1145/2940296 (cit. on p. 138).

[362] André Joaquim, Miguel L. Pardal and Miguel Correia. 2017. Vulnerability-tolerant transport layer security. In *21st International Conference on Principles of Distributed Systems, OPODIS 2017, Lisbon, Portugal, December 18-20, 2017* (LIPIcs). Vol. 95. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 28:1–28:16. DOI: 10.4230/LIPICS.OPODIS.2017.28 (cit. on p. 138).

[363] Christian Göttel, Pascal Felber and Valerio Schiavoni. 2019. Developing secure services for IoT with OP-TEE: a first look at performance and usability. In *Distributed Applications and Interoperable Systems - 19th IFIP WG 6.1 International Conference, DAIS 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings* (Lecture Notes in Computer Science). Vol. 11534. Springer, 170–178. DOI: 10.1007/978-3-030-22496-7_11 (cit. on p. 138).

[364] Pierre-Louis Aublin, Florian Kelbert, Dan O'Keffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers and Peter Pietzuch. 2017. TaLoS: Secure and transparent TLS termination inside SGX enclaves. Tech. rep. Department of Computing, Imperial College London. https://www.doc.ic.ac.uk/research/technicalreports/2017/DTRS17-5.pdf (cit. on p. 139).

[365] [SW] Apache Software Foundation, Apache Kafka, an open-source distributed event streaming platform version 3.8.0, 29 July 2024. Retrieved 4 Sept. 2024 from URL: https://kafka.apache.org/ (cit. on p. 139).

[366] [SW] VMware LLC, RabbitMQ, one broker to queue them all version 3.13.7, 26 Aug. 2024. Retrieved 4 Sept. 2024 from URL: https://www.rabbitmq.com/ (cit. on p. 139).

[367] Roger A. Light. 2017. Mosquitto: server and client implementation of the MQTT protocol. *J. Open Source Softw.*, 2, 13, 265. DOI: 10.21105/JOSS.00265 (cit. on pp. 139, 147).

[368] Amazon. 2010. Simple Notification Service, a fully managed pub/sub service for A2A and A2P messaging. (9 Apr. 2010). Retrieved 4 Sept. 2024 from https://aws.amazon.com/sns/ (cit. on p. 139).

[369] Chenxi Wang, Antonio Carzaniga, David Evans and Alexander L Wolf. 2002. Security issues and requirements for internet-scale publish-subscribe systems. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*. IEEE, 3940–3947. DOI: 10.1109/HICSS.2002.994531 (cit. on p. 140).

[370] Mihaela Ion, Giovanni Russello and Bruno Crispo. 2012. Design and implementation of a confidentiality and access control solution for publish/subscribe systems. *Comput. Networks*, 56, 7, 2014–2037. DOI: 10.1016/j.comnet.2012.02.013 (cit. on p. 140).

[371] Mohamed Nabeel, Stefan Appel, Elisa Bertino and Alejandro P. Buchmann. 2013. Privacy preserving context aware publish subscribe systems. In *Network and System Security - 7th International Conference, NSS 2013, Madrid, Spain, June 3-4, 2013. Proceedings* (Lecture Notes in Computer Science). Vol. 7873. Springer, 465–478. DOI: 10.1007/978-3-642-38631-2_34 (cit. on p. 140).

[372] Raphaël Barazzutti, Pascal Felber, Hugues Mercier, Emanuel Onica and Etienne Rivière. 2017. Efficient and confidentiality-preserving content-based publish/subscribe with prefiltering. *IEEE Trans. Dependable Secur. Comput.*, 14, 3, 308–325. DOI: 10.1109/TDSC.2015.2449831 (cit. on p. 140).

[373] Cristian Borcea, Arnab Deb Gupta, Yuriy Polyakov, Kurt Rohloff and Gerard W. Ryan. 2017. PICADOR: end-to-end encrypted publish-subscribe information distribution with proxy re-encryption. *Future Gener. Comput. Syst.*, 71, 177–191. DOI: 10.1016/j.future.2016.10.013 (cit. on p. 140).

[374] Lukas Malina, Gautam Srivastava, Petr Dzurenda, Jan Hajny and Radek Fujdiak. 2019. A secure publish/subscribe protocol for internet of things. In *Proceedings of the 14th International Conference on Availability, Reliability and Security, ARES 2019, Canterbury, UK, August 26-29, 2019*. ACM, 75:1–75:10. DOI: 10.1145/3339252.3340503 (cit. on p. 140).

[375] Sarah Abdelwahab Gaballah, Christoph Coijanovic, Thorsten Strufe and Max Mühlhäuser. 2021. 2PPS – publish/subscribe with provable privacy. In *40th International Symposium on Reliable Distributed Systems, SRDS 2021, Chicago, IL, USA, September 20-23, 2021*. IEEE, 198–209. DOI: 10.1109/SRDS53918.2021.00028 (cit. on p. 140).

[376] Stefano Berlato, Umberto Morelli, Roberto Carbone and Silvio Ranise. 2022. End-to-end protection of IoT communications through cryptographic enforcement of access control policies. In *Data and Applications Security and Privacy XXXVI - 36th Annual IFIP WG 11.3 Conference, DBSec 2022, Newark, NJ, USA, July 18-20, 2022, Proceedings* (Lecture Notes in Computer Science). Vol. 13383. Springer, 236–255. DOI: 10.1007/978-3-031-10684-2_14 (cit. on p. 140).

[377] Rafael Pires, Marcelo Pasin, Pascal Felber and Christof Fetzer. 2016. Secure content-based routing using Intel software guard extensions. In *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016*. ACM, 10. https://doi.org/10.1145/2988336.2988346 (cit. on p. 140).

[378] Sergei Arnautov, Andrey Brito, Pascal Felber, Christof Fetzer, Franz Gregor, Robert Krahn, Wojciech Ozga, André Martin, Valerio Schiavoni, Fábio Silva, Marcus Tenorio and Nikolaus Thummel. 2018. PubSub-SGX: exploiting trusted execution environments for privacy-preserving publish/subscribe systems. In *37th IEEE Symposium on Reliable Distributed Systems, SRDS 2018, Salvador, Brazil, October 2-5, 2018*. IEEE Computer Society, 123–132. DOI: 10.1109/SRDS.2018.00023 (cit. on p. 140).

[379] Shuran Wang, Dahan Pan, Runhan Feng and Yuanyuan Zhang. 2021. Magikcube: securing cross-domain publish/subscribe systems with enclave. In *20th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2021, Shenyang, China, October 20-22, 2021*. IEEE, 147–154. DOI: 10.1109/TrustCom53373.2021.00037 (cit. on p. 140).

[380] Jinglei Pei, Yuyang Shi, Qingling Feng, Ruisheng Shi, Lina Lan, Shui Yu, Jinqiao Shi and Zhaofeng Ma. 2023. An efficient confidentiality protection solution for pub/sub system. *Cybersecur.*, 6, 1, 34. DOI: 10.1186/s42400-023-00165-w (cit. on p. 140).

[381] Carlos Segarra, Ricard Delgado-Gonzalo and Valerio Schiavoni. 2020. MQT-TZ: hardening IoT brokers using ARM TrustZone : (practical experience report). In *International Symposium on Reliable Distributed Systems, SRDS 2020, Shanghai, China, September 21-24, 2020*. IEEE, 256–265. DOI: 10.1109/SRDS51746.2020.00033 (cit. on p. 140).

[382] N. Asokan, Valtteri Niemi and Kaisa Nyberg. 2003. Man-in-the-middle in tunnelled authentication protocols. In *Security Protocols, 11th International Workshop, Cambridge, UK, April 2-4, 2003, Revised Selected Papers* (Lecture Notes in Computer Science). Vol. 3364. Springer, 28–41. DOI: 10.1007/11542322_6 (cit. on p. 141).

[383] Kenneth A. Goldman, Ronald Perez and Reiner Sailer. 2006. Linking remote attestation to secure tunnel endpoints. In *Proceedings of the 1st ACM Workshop on Scalable Trusted Computing, STC 2006, Alexandria, VA, USA, November 3, 2006*. ACM, 21–24. DOI: 10.1145/1179474.1179481 (cit. on p. 142).

[384] Frederic Stumpf, Omid Tafreschi, Patrick Röder and Claudia Eckert. 2006. A robust integrity reporting protocol for remote attestation. In *Second Workshop on Advances in Trusted Computing (WATC '06 Fall)*. Tokyo, Japan, (Nov. 2006) (cit. on p. 142).

[385] Yacine Gasmi, Ahmad-Reza Sadeghi, Patrick Stewin, Martin Unger and N. Asokan. 2007. Beyond secure channels. In *Proceedings of the 2nd ACM Workshop on Scalable Trusted Computing, STC 2007, Alexandria, VA, USA, November 2, 2007*. ACM, 30–40. DOI: 10.1145/1314354.1314363 (cit. on p. 142).

[386] Frederic Stumpf, Andreas Fuchs, Stefan Katzenbeisser and Claudia Eckert. 2008. Improving the scalability of platform attestation. In *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing, STC 2008, Alexandria, VA, USA, October 31, 2008*. ACM, 1–10. DOI: 10.1145/1456455.1456457 (cit. on p. 142).

[387] Frederik Armknecht, Yacine Gasmi, Ahmad-Reza Sadeghi, Patrick Stewin, Martin Unger, Gianluca Ramunno and Davide Vernizzi. 2008. An efficient implementation of trusted channels based on openssl. In *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing, STC 2008, Alexandria, VA, USA, October 31, 2008*. ACM, 41–50. DOI: 10.1145/1456455.1456462 (cit. on p. 142).

[388] Yue Yu, Huaimin Wang, Bo Liu and Gang Yin. 2013. A trusted remote attestation model based on trusted computing. In *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2013 / 11th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA-13 / 12th IEEE International Conference on Ubiquitous Computing and Communications, IUCC-2013, Melbourne, Australia, July 16-18, 2013*. IEEE Computer Society, 1504–1509. DOI: 10.1109/TrustCom.2013.183 (cit. on p. 142).

[389] Andrew James Paverd. 2015. *Enhancing communication privacy using trustworthy remote entities*. PhD thesis. University of Oxford (cit. on p. 142).

[390] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor and Adrian Perrig. 2010. TrustVisor: efficient TCB reduction and attestation. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 143–158. DOI: 10.1109/SP.2010.17 (cit. on p. 142).

[391] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel and Bryan Parno. 2017. Komodo: using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 287–305. DOI: 10.1145/3132747.3132782 (cit. on p. 142).

[392] Trusted Computing Group. 2021. *DICE Attestation Architecture*. Version 1.00 r23. (1 Mar. 2021). 36 pp. Retrieved 6 Sept. 2024 from https://trustedcomputinggroup.org/wp-content/uploads/DICE-Attestation-Architecture-r23-final.pdf (cit. on p. 142).

[393] Sébanjila Kevin Bukasa, Ronan Lashermes, Hélène Le Bouder, Jean-Louis Lanet and Axel Legay. 2017. How TrustZone could be bypassed: side-channel attacks on a modern system-on-chip. In *Information Security Theory and Practice, 11th IFIP WG 11.2 International Conference, WISTP 2017, Heraklion, Crete, Greece, September 28-29, 2017, Proceedings* (Lecture Notes in Computer Science). Vol. 10741. Springer, 93–109. DOI: 10.1007/978-3-319-93524-9_6 (cit. on p. 143).

[394] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis and Fabian Monrose. 2019. The severest of them all: inference attacks against secure virtual enclaves. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, AsiaCCS 2019, Auckland, New Zealand, July 09-12, 2019*. ACM, 73–85. DOI: 10.1145/3321705.3329820 (cit. on p. 143).

[395] Shufan Fei, Zheng Yan, Wenxiu Ding and Haomeng Xie. 2022. Security vulnerabilities of SGX and countermeasures: A survey. *ACM Comput. Surv.*, 54, 6, 126:1–126:36. DOI: 10.1145/3456631 (cit. on p. 143).

[396] A. K. M. Mubashwir Alam and Keke Chen. 2023. Making your program oblivious: A comparative study for side-channel-safe confidential computing. In *16th IEEE International Conference on Cloud Computing, CLOUD 2023, Chicago, IL, USA, July 2-8, 2023*. IEEE, 282–289. DOI: 10.1109/CLOUD60044.2023.00040 (cit. on p. 143).

[397] Peter Saint-Andre and Jeff Hodges. 2011. Representation and verification of domain-based application service identity within internet public key infrastructure using X.509 (PKIX) certificates in the context of transport layer security (TLS). RFC 6125. (Mar. 2011). DOI: 10.17487/RFC6125 (cit. on p. 146).

[398] Eric Rescorla. 2010. Keying material exporters for transport layer security (TLS). RFC 5705. (Mar. 2010). DOI: 10.17487/RFC5705 (cit. on p. 147).

[399] Eric Rescorla, Kazuho Oku, Nick Sullivan and Christopher A. Wood. 2023. TLS Encrypted Client Hello. Internet-Draft draft-ietf-tls-esni-16. Work in Progress. Internet Engineering Task Force, (Apr. 2023). 48 pp. https://datatracker.ietf.org/doc/draft-ietf-tls-esni/16/ (cit. on p. 147).

[400] Robin Vassantlal, Eduardo Alchieri, Bernardo Ferreira and Alysson Bessani. 2022. COBRA: dynamic proactive secret sharing for confidential BFT services. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 1335–1353. DOI: 10.1109/SP46214.2022.9833658 (cit. on p. 148).

[401] Sébastien Vaucher, Valerio Schiavoni and Pascal Felber. 2018. Short paper: Stress-SGX: load and stress your enclaves for fun and profit. In *Networked Systems - 6th International Conference, NETYS 2018, Essaouira, Morocco, May 9-11, 2018, Revised Selected Papers* (Lecture Notes in Computer Science). Vol. 11028. Springer, 358–363. DOI: 10.1007/978-3-030-05529-5_24 (cit. on p. 151).

[402] Wasmer. 2023. WASIX, a long term stabilization and support of the existing WASI ABI plus additional non-invasive syscall extensions. (23 May 2023). Retrieved 12 Sept. 2024 from https://wasmer.io/posts/announcing-wasix (cit. on p. 159).

[403] Arjun Ramesh, Tianshu Huang, Ben L. Titzer and Anthony Rowe. 2023. Stop hiding the sharp knives: the WebAssembly Linux interface. *CoRR*, abs/2312.03858. arXiv: 2312.03858. DOI: 10.48550/ARXIV.2312.03858 (cit. on p. 159).

[404] Randall Munroe. 2011. How standards proliferate. (22 July 2011). Retrieved 12 Sept. 2024 from https://xkcd.com/927/ (cit. on p. 159).

[405] Jianping Zhu, Rui Hou, XiaoFeng Wang, Wenhao Wang, Jiangfeng Cao, Boyan Zhao, Zhongpu Wang, Yuhui Zhang, Jiameng Ying, Lixin Zhang and Dan Meng. 2020. Enabling rack-scale confidential computing using heterogeneous trusted execution environment. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1450–1465. DOI: 10.1109/SP40000.2020.00054 (cit. on p. 159).

[406] Yunjie Deng, Chenxu Wang, Shunchang Yu, Shiqing Liu, Zhenyu Ning, Kevin Leach, Jin Li, Shoumeng Yan, Zhengyu He, Jiannong Cao and Fengwei Zhang. 2022. StrongBox: a GPU TEE on Arm endpoints. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*. ACM, 769–783. DOI: 10.1145/3548606.3560627 (cit. on p. 159).

[407] Xiaolong Wu, Dave Jing Tian and Chung Hwan Kim. 2023. Building GPU TEEs using CPU secure enclaves with GEVisor. In *Proceedings of the 2023 ACM Symposium on Cloud Computing, SoCC 2023, Santa Cruz, CA, USA, 30 October 2023 - 1 November 2023*. ACM, 249–264. DOI: 10.1145/3620678.3624659 (cit. on p. 159).

[408] Florian Tramèr and Dan Boneh. 2019. Slalom: fast, verifiable and private execution of neural networks in trusted hardware. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=rJVorjCcKQ (cit. on p. 160).

[409] Jean-Baptiste Truong, William Gallagher, Tian Guo and Robert J. Walls. 2021. Memory-efficient deep learning inference in trusted execution environments. In *IEEE International Conference on Cloud Engineering, IC2E 2021, San Francisco, CA, USA, October 4-8, 2021*. IEEE, 161–167. DOI: 10.1109/IC2E52221.2021.00031 (cit. on p. 160).

[410] Z. Zhang, C. Gong, Y. Cai, Y. Yuan, B. Liu, D. Li, Y. Guo and X. Chen. 2024. No privacy left outside: on the (in-)security of TEE-shielded DNN partition for on-device ML. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, (May 2024), 3327–3345. DOI: 10.1109/SP54263.2024.00052 (cit. on p. 160).

[411] Hanieh Hashemi, Yongqin Wang and Murali Annavaram. 2021. DarKnight: an accelerated framework for privacy and integrity preserving deep learning using trusted hardware. In *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021*. ACM, 212–224. DOI: 10.1145/3466752.3480112 (cit. on p. 160).

[412] Ke Xia, Yukui Luo, Xiaolin Xu and Sheng Wei. 2021. SGX-FPGA: trusted execution environment for CPU-FPGA heterogeneous architecture. In *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*. IEEE, 301–306. DOI: 10.1109/DAC18074.2021.9586207 (cit. on p. 160).

[413] Ardhi Wiratama Baskara Yudha, Jake Meyer, Shougang Yuan, Huiyang Zhou and Yan Solihin. 2022. LITE: a low-cost practical inter-operable GPU TEE. In *ICS '22: 2022 International Conference on Supercomputing, Virtual Event, June 28 - 30, 2022*. ACM, 7:1–7:13. DOI: 10.1145/3524059.3532361 (cit. on p. 160).

[414] NVIDIA Corporation. 2023. Confidential Compute on NVIDIA Hopper H100. White paper WP-11459-001. Version 1.0. (25 July 2023). 32 pp. Retrieved 13 Sept. 2024 from https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/HCC-Whitepaper-v1.0.pdf (cit. on p. 160).

[415] Jianwei Zhu, Hang Yin and Shunfan Zhou. 2024. Confidential computing on NVIDIA H100 GPU: a performance benchmark study. (2024). https://arxiv.org/abs/2409.03992 arXiv: 2409.03992 [cs.DC] (cit. on p. 160).

[416] VMware LLC. 2012. Impact of Enhanced vMotion Compatibility on Application Performance. White paper EN-000988-00. (10 Aug. 2012). 12 pp. Retrieved 11 Sept. 2024 from https://www.vmware.com/docs/vmware-vsphere-evc-performance-white-paper (cit. on p. 161).

[417] Kiryong Ha, Yoshihisa Abe, Thomas Eiszler, Zhuo Chen, Wenlu Hu, Brandon Amos, Rohit Upadhyaya, Padmanabhan Pillai and Mahadev Satyanarayanan. 2017. You can teach elephants to dance: agile VM handoff for edge computing. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing, San Jose / Silicon Valley, SEC 2017, CA, USA, October 12-14, 2017*. ACM, 12:1–12:14. DOI: 10.1145/3132211.3134453 (cit. on p. 161).

[418] Andrew Machen, Shiqiang Wang, Kin K. Leung, Bong Jun Ko and Theodoros Salonidis. 2018. Live service migration in mobile edge clouds. *IEEE Wirel. Commun.*, 25, 1, 140–147. DOI: 10.1109/MWC.2017.1700011 (cit. on p. 161).

[419] Adam Ruprecht, Danny Jones, Dmitry Shiraev, Greg Harmon, Maya Spivak, Michael Krebs, Miche Baker-Harvey and Tyler Sanderson. 2018. VM live migration at scale. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2018, Williamsburg, VA, USA, March 25-25, 2018*. ACM, 45–56. DOI: 10.1145/3186411.3186415 (cit. on p. 161).

[420] Jinyu Gu, Zhichao Hua, Yubin Xia, Haibo Chen, Binyu Zang, Haibing Guan and Jinming Li. 2017. Secure live migration of SGX enclaves on untrusted cloud. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*. IEEE Computer Society, 225–236. DOI: 10.1109/DSN.2017.37 (cit. on p. 161).

[421] Fritz Alder, Arseny Kurnikov, Andrew Paverd and N. Asokan. 2018. Migrating SGX enclaves with persistent state. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*. IEEE Computer Society, 195–206. DOI: 10.1109/DSN.2018.00031 (cit. on p. 161).

[422] Jianqiang Wang, Pouya Mahmoody, Ferdinand Brasser, Patrick Jauernig, Ahmad-Reza Sadeghi, Donghui Yu, Dahan Pan and Yuanyuan Zhang. 2022. VirTEE: a full backward-compatible TEE with native live migration and secure I/O. In *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022*. ACM, 241–246. DOI: 10.1145/3489517.3530436 (cit. on p. 161).

[423] Fang-Jie Yang, Jian-Lin Li, Kaiwen Xue and Shih-Wei Li. 2024. Designing and implementing live migration support for Arm-based confidential VMs. In *Proceedings of the 15th ACM SIGOPS Asia-Pacific Workshop on Systems, APSys 2024, Kyoto, Japan, September 4-5, 2024*. ACM, 53–59. DOI: 10.1145/3678015.3680488 (cit. on p. 161).

[424] Jian-Lin Li and Shih-Wei Li. 2024. Performance implications of sev virtual machine live migration. In *19th Workshop on Virtualization in High-Performance Cloud Computing, VHPC 2024, Madrid, Spain, Aug 27, 2024* (cit. on p. 161).

[425] Hyuk-Jin Jeong, Chang Hyun Shin, Kwang Yong Shin, Hyeon-Jae Lee and Soo-Mook Moon. 2019. Seamless offloading of web app computations from mobile device to edge clouds via HTML5 web worker migration. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 38–49. DOI: 10.1145/3357223.3362735 (cit. on p. 162).

[426] Mohammed Nurul Hoque and Khaled A. Harras. 2021. Nomad: cross-platform computational offloading and migration in Femtoclouds using WebAssembly. In *IEEE International Conference on Cloud Engineering, IC2E 2021, San Francisco, CA, USA, October 4-8, 2021*. IEEE, 168–178. DOI: 10.1109/IC2E52221.2021.00032 (cit. on p. 162).

[427] Daigo Fujii, Katsuya Matsubara and Yuki Nakata. 2024. Stateful VM migration among heterogeneous WebAssembly runtimes for efficient edge-cloud collaborations. In *Proceedings of the 7th International Workshop on Edge Systems, Analytics and Networking, EdgeSys 2024, Athens, Greece, 22 April 2024*. ACM, 19–24. DOI: 10.1145/3642968.3654816 (cit. on p. 162).

[428] Manuel Nieke, Lennart Almstedt and Rüdiger Kapitza. 2021. Edgedancer: secure mobile WebAssembly services on the edge. In *EdgeSys@EuroSys 2021: 4th International Workshop on Edge Systems, Analytics and Networking, Online Event, United Kingdom, April 26, 2021*. ACM, 13–18. DOI: 10.1145/3434770.3459731 (cit. on p. 162).

[429] Vasile Adrian Bogdan Pop, Arto Niemi, Valentin Manea, Antti Rusanen and Jan-Erik Ekberg. 2022. Towards securely migrating WebAssembly enclaves. In *EuroSec@EUROSYS 2022: Proceedings of the 15th European Workshop on Systems Security, Rennes, France, April 5-8, 2022*. ACM, 43–49. DOI: 10.1145/3517208.3523755 (cit. on p. 162).

[430] Steven Osman, Dinesh Subhraveti, Gong Su and Jason Nieh. 2002. The design and implementation of Zap: a system for migrating computing environments. In *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*. USENIX Association. http://www.usenix.org/events/osdi02/tech/osman.html (cit. on p. 162).

[431] Oren Laadan, Dan B. Phung and Jason Nieh. 2005. Transparent checkpoint-restart of distributed applications on commodity clusters. In *2005 IEEE International Conference on Cluster Computing (CLUSTER 2005), September 26 - 30, 2005, Boston, Massachusetts, USA*. IEEE Computer Society, 1–13. DOI: 10.1109/CLUSTR.2005.347039 (cit. on p. 162).

[432] Ifeanyi P. Egwutuoha, David Levy, Bran Selic and Shiping Chen. 2013. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *J. Supercomput.*, 65, 3, 1302–1326. DOI: 10.1007/S11227-013-0884-0 (cit. on p. 162).