Université de Neuchâtel
Faculté des sciences
Institut d'informatique

Distributed systems and trusted execution environments: Trade-offs and challenges

par

Rafael Pereira Pires

Thèse

présentée à la Faculté des sciences

pour l'obtention du grade de Docteur ès sciences

Acceptée sur proposition du jury:

Prof. Pascal Felber, directeur de thèse
Université de Neuchâtel, Suisse

Prof. Marcelo Pasin, codirecteur de thèse
Haute école Arc, Suisse

Dr. Valerio Schiavoni
Université de Neuchâtel, Suisse

Prof. Daniel Lucani
Aarhus University, Danemark

Prof. Gaël Thomas
Telecom SudParis, France

Soutenue le 3 décembre 2019

UNIVERSITÉ DE NEUCHÂTEL

Faculté des Sciences
Secrétariat-décanat de Faculté
Rue Emile-Argand 11
2000 Neuchâtel – Suisse
Tél : + 41 (0)32 718 21 00
E-mail : secretariat.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 Rafael PEREIRA PIRES

Titre :

"Distributed systems and trusted execution environments: Trade-offs and challenges"

sur le rapport des membres du jury composé comme suit :

- Prof. Pascal Felber, directeur de thèse, Université de Neuchâtel, Suisse
- Assoc. Prof. Daniel Enrique Lucani Rötter, Aarhus University, Danemark
- Prof. Gaël Thomas, Telecom SudParis, France
- Prof. Marcelo Pasin, Haute école Arc, Neuchâtel, Suisse
- Dr Valerio Schiavoni, Université de Neuchâtel, Suisse

Neuchâtel, le 5 décembre 2019

Le Doyen, Prof. P. Felber

Abstract

Security and privacy concerns in computer systems have grown in importance with the ubiquity of connected devices. Additionally, cloud computing boosts such distress as private data is stored and processed in multi-tenant infrastructure providers. In recent years, trusted execution environments (TEEs) have caught the attention of scientific and industry communities as they became largely available in user- and server-class machines.

TEEs provide security guarantees based on cryptographic constructs built in hardware. Since silicon chips are difficult to probe or reverse engineer, they can offer stronger protection against remote or even physical attacks when compared to their software counterparts. Intel software guard extensions (SGX), in particular, implements powerful mechanisms that can shield sensitive data even from privileged users with full control of system software.

Designing secure distributed systems is a notably daunting task, since they involve many coordinated processes running in geographically-distant nodes, therefore having numerous points of attack. In this work, we essentially explore some of these challenges by using Intel SGX as a crucial tool. We do so by designing and experimentally evaluating several elementary systems ranging from communication and processing middleware to a peer-to-peer privacy-preserving solution.

We start with support systems that naturally fit cloud deployment scenarios, namely content-based routing, batching and stream processing frameworks. Our communication middleware protects the most critical stage of matching subscriptions against publications inside secure enclaves and achieves substantial performance gains in comparison to traditional software-based equivalents. The processing platforms, in turn, receive encrypted data and code to be executed within the trusted environment. Our prototypes are then used to analyse the manifested memory usage issues intrinsic to SGX.

Next, we aim at protecting very sensitive data: cryptographic keys. By leveraging TEEs, we design protocols for group data sharing that have lower computational complexity than legacy methods. As a bonus, our proposals allow large savings on metadata volume and processing time of cryptographic operations, all with equivalent security guarantees.

Finally, we focus our attention on privacy-preserving systems. After all, users cannot modify some existing systems like web-search engines, and the providers of these services may keep individual profiles containing sensitive private information about them. We aim at achieving indistinguishability and unlinkability properties by employing techniques like sensitivity analysis, query obfuscation and leveraging relay nodes. Our evaluation shows that we propose the most robust system in comparison to existing solutions with regard to user re-identification rates and results' accuracy in a scalable way.

All in all, this thesis proposes new mechanisms that take advantage of TEEs for distributed system architectures. We show through an empirical approach on top of Intel SGX what are the trade-offs of distinct designs applied to distributed communication and processing, cryptographic protocols and private web search.

Keywords: security, privacy, TEE, SGX, distributed systems, communication, processing, cryptographic protocols, web search.

Résumé

Les problèmes de sécurité et de confidentialité des systèmes informatiques ont pris de l'importance avec l'omniprésence des périphériques connectés. En outre, l'informatique en nuage accroît cette détresse, car les données privées sont stockées et traitées par des fournisseurs d'infrastructure hébergeant de multiples locataires. Ces dernières années, les environnements d'exécution de confiance, ou TEE (« trusted execution environments ») ont attiré l'attention des communautés scientifiques et industrielles, car ils sont devenus largement disponibles sur des machines de type utilisateur et serveur.

Les TEE fournissent des garanties de sécurité basées sur des constructions cryptographiques intégrées au matériel. Les puces de silicium étant difficiles à sonder ou étudier par ingénierie inverse, elles offrent une protection renforcée contre les attaques distantes, voire physiques, par rapport à leurs homologues logicielles. En particulier, les extensions de protection logicielle Intel SGX (« software guard extensions ») implémentent de puissants mécanismes qui peuvent protéger les données sensibles même contre les utilisateurs privilégiés disposant du contrôle total du logiciel de système.

La conception de systèmes repartis sécurisés est une tâche particulièrement ardue, car ils impliquent de nombreux processus coordonnés exécutés dans des nœuds géographiquement distants, ce qui entraîne de nombreux points d'attaque. Dans ce travail, nous explorons essentiellement certains de ces défis en utilisant Intel SGX comme pierre angulaire. Nous le faisons en concevant et en évaluant de manière expérimentale plusieurs systèmes élémentaires allant du logiciel médiateur de communication et de traitement à une solution de protection de la confidentialité pair-à-pair.

Nous commençons par des systèmes de support qui s'adaptent naturellement aux scénarios de déploiement dans le *cloud*, à savoir : des infrastructures de routage en fonction du contenu, de traitement par lots et de traitement de flux. Notre logiciel médiateur de communication protège la phase de mise en correspondance d'abonnements avec des publications, qui est la plus critique, en l'effectuant à l'intérieur d'enclaves sécurisées. Il permet des gains substantiels en performance par rapport aux équivalents traditionnels basés sur des logiciels. Les plates-formes de traitement reçoivent à leur tour des données chiffrées et du code à exécuter dans l'environnement sécurisé. Nos prototypes sont ensuite utilisés pour analyser les problèmes d'utilisation de mémoire qui sont inhérents à SGX.

Nous visons ensuite à protéger des données très sensibles : les clés cryptographiques. En utilisant les TEE, nous concevons des protocoles pour le partage de données de groupe présentant une complexité de calcul inférieure à celle des méthodes traditionnelles. De plus, nos propositions permettent d'importantes économies quant au volume de méta-données produites et au temps de traitement des opérations cryptographiques, le tout avec des garanties de sécurité équivalentes.

Enfin, nous concentrons notre attention sur les systèmes préservant la confidentialité. Après tout, les utilisateurs ne peuvent pas modifier certains systèmes existants, tels que les moteurs de recherche Web. Les fournisseurs de ces services peuvent conserver des profils individuels contenant des informations confidentielles les concernant. Nous visons à obtenir des propriétés d'indiscernabilité et indissociabilité en utilisant des techniques telles que l'analyse de sensibilité, l'obscurcissement de requêtes et l'utilisation de nœuds relais. Notre évaluation montre que nous proposons le système le plus robuste par rapport aux solutions existantes en ce qui concerne les taux de ré-identification des utilisateurs et la précision des résultats de manière évolutive.

Dans sa globalité, cette thèse propose de nouveaux mécanismes tirant parti des TEE pour les architectures de systèmes repartis. Nous montrons par approche empirique au-dessus de Intel SGX quels sont les compromis entre plusieurs modèles de conception distincts appliqués à la communication et au traitement répartis, aux protocoles cryptographiques et à la recherche privée sur le Web.

Mots clés : sécurité, confidentialité, TEE, SGX, systèmes repartis, communication, traitement, protocoles cryptographiques, recherche sur le Web.

To you, who is reading this.

Acknowledgements

Doing a PhD is not trivial, as a good friend might say. We work hard, *on est schlagés du galetas*, at any time, at weekends and on holidays. And yet our masterpieces are rejected by evil reviewers. Nice ones accept them though, and that is amazing. We celebrate and we travel. I personally presented my work in meetings, conferences and seminars that took place in Switzerland, Denmark, Romania, France, Germany, Spain, Italy, Luxembourg and the USA. It is the joy, but then it is gone. Like addicts, we react by working hard for the next one. C'est la vie ! In spite of pros and cons, a PhD is made by people, many people. And I dedicate this work to them.

First and foremost, I would like to express my gratitude to my advisors, Prof. Pascal Felber and Prof. Marcelo Pasin. Pascal provided me with all the support I needed. His cleverness in conceiving and assigning tasks and his amazing network of partners were crucial for this work's accomplishments. Apart from that, he excels in every kind of sport. We had a remarkable work meeting while hiking down a mountain in Sinaia, Romania, and went a few times for indoor climbing during lunch breaks. Thanks for everything!

Pasin was responsible for my coming to Neuchâtel. Once I arrived, he helped me with administrative stuff, gave me private lectures about computer science subjects (my years in industry made me somewhat rusty) and made excellent Brazilian barbecues at his place. Along the way, we interacted in several technical discussions, project deliverables and also in career counselling. Muito obrigado!

Many thanks to the external committee members Prof. Daniel Lucani and Prof. Gaël Thomas, who provided valuable comments and suggestions for improving this manuscript. I am also really grateful to all the co-authors with whom I had the privilege to work. Special thanks to Dr. Emanuel Onica, Dr. Stefan Contiu and Dr. Sonia Ben Mokhtar.

Dr. Valerio Schiavoni, who was also a committee member, provided me valuable comments. We worked together in part of the results of this thesis. He is skilled in subtle task assignment and impressively follows up the work of virtually all PhD students in the group. But not only: he goes along by working hard on experiments and writing, even if we happen to be in a casino-hotel in Las Vegas at 3 a.m. Grazie mille!

Thanks to all colleagues in the Department, including but not limited to Heverson Ribeiro, Raluca Halalai, Aurélien Havet, Sukanya Nath, Laurent Hayez and Catherine Ikae. Many thanks to Roberta Barbi and her discreet laugh, Andrei Lapin and his rollers, Yarco Hayduk and his colourful attire, Mascha Kurpicz and her language abilities, Mirco Kocher and his taste for mushrooms, Maria Carpen-Amarie and her strong opinions, Raziel Gómez and his salsa, Raphaël Barazzutti and his washing machine, Veronica Estrada and her *sangre latino*, Christian Göttel and his motorbike, Lars Nielsen and his denmarkness, Nils Schaetti and his silence, Isabelly Rocha and her cross-fit, Rémi Dulong and his musical instruments, and to Peterson Yuhala and his kebabs. Every lunch, board games evenings and gatherings in bars, street festivals or in someone's place helped to build a friendly atmosphere. This had the power to attenuate tensions of any kind (including pre-deadline ones) and gave strength and motivation to go to work every day.

Merci au personnel de l'Institut d'informatique, y compris les secrétaires Émilie Auclair et Debora Mendes, les concierges Álvaro et Afrim Sadiku, et aux maîtres-assistants et professeurs Peter Kropf, Jacques Savoy, Alain Sandoz, Hervé Sanglard, Hugues Mercier et Lorenzo Leonini. C'était toujours agréable de partager avec vous des moments aux soupers de fin d'année, à la torée neuchâteloise en été et dans l'institut.

Un grand merci à Dorian Burihabwa, qui a relu ce texte et suggéré des corrections. D'ailleurs, malgré ses recommandations pour les mauvais films, j'ai bien apprécié ses visites quotidiennes dans notre bureau

(Séb et moi) et sa disponibilité en étant toujours prêt à boire un verre en soirée ou un brunch les week-ends.

Je remercie en particulier mon pote et compagnon de bureau, Sébastien Vaucher. On a bossé ensemble sur quelques projets, et il m'a appris pas mal de choses. Ses connaissances sur la Suisse et sur tous les réglements de n'importe quoi sont impressionnantes. Mais c'est surtout sa bonne humeur qui m'a fait prendre plaisir de passer mon temps au bureau. Merci bien !

Sébastien et Rémi m'ont appris un tas d'expressions francophones. Quelques-unes restent pour moi un mystère et boule de gomme, mais au moins je sais maintenant qu'il ne faut jamais pousser Mémé dans les orties, malgré le fait que parfois on tape d'abord et ensuite on interroge. En fait, Dorian, Rémi et Sébastien ont été les responsables du déclenchement de mon français. Merci les gars !

Meu muito obrigado vai aos amigos do Brasil que vieram me visitar em Neuchâtel: Edmar Araújo, Ana e Cesar Guerra; e aos que me acolheram em Porto Alegre (e em Paris) — Marcelo Neves, Mônica Marcuzzo e Rubens Belusso —, em Floripa — Luiza Delpretto e Edmar —, em Miami — Roberta e Juliano Reckziegel —, e em Barcelona/Melbourne — Bárbara Cardoso e Danillo Santos. Agradeço também a Natália Bortolás, que fez parte dessa jornada.

Enfim, agradeço imensamente à minha família. Meus pais, Jorge e Rejane, irmãos Gaspar e Bárbara, sobrinhos e sobrinha, avós, tias e tios, cunhada e cunhado, primas e primos. O incentivo, suporte e carinho de vocês são as bases sobre as quais todas as minhas conquistas são construídas.

Well, until here I wrote in 6 languages, cited 11 countries and 57 names (non-exhaustive list), who in turn are citizens of 20 countries. It is a good indication that this work is the result of a multi-cultural environment and had many, many contributors. Vielen Dank an alle.

Contents

Abstract **v**

Résumé **vii**

Acknowledgements **xi**

List of acronyms **xv**

List of figures **xviii**

1 Introduction **1**
1.1 Context and motivation . . . 1
1.2 Contributions . . . 2
1.3 Outline . . . 3

2 Background and related work **5**
2.1 Trusted execution environments . . . 5
2.1.1 Trusted platform modules . . . 7
2.1.2 ARM TrustZone . . . 7
2.1.3 AMD secure encrypted virtualisation (SEV) . . . 8
2.2 Intel SGX . . . 9
2.2.1 Memory protection . . . 11
2.2.2 Limitations and performance implications . . . 12
2.2.3 Enclave signing and attestation . . . 14
2.2.4 Sealing . . . 15
2.2.5 Platform services . . . 15
2.2.6 Vulnerabilities . . . 16
2.3 SecureCloud project . . . 17
2.4 Communication and data processing . . . 18
2.4.1 Secure publish/subscribe . . . 18
2.4.2 Batch processing . . . 19
2.4.3 Stream processing . . . 20
2.5 Data sharing . . . 20
2.6 Privacy assurance . . . 23
2.7 Summary . . . 27

3 Communication and processing **29**
3.1 Communication . . . 29
3.1.1 Secure content-based routing . . . 30
3.2 Processing . . . 35
3.2.1 Lua within enclaves . . . 36
3.2.2 Lightweight MapReduce . . . 37
3.2.3 SecureStreams . . . 45
3.3 Summary . . . 50

4 Group communication and data sharing **53**
4.1 Cryptographic group access control . . . 53
4.1.1 IBBE-SGX . . . 55

4.2 Anonymous file sharing 65
4.2.1 A-Sky 66
4.3 Summary 75

5 Privacy enforcement **77**
5.1 Private web search 77
5.1.1 X-Search 78
5.2 TEE in the client-side 84
5.2.1 Cyclosa 85
5.3 Summary 93

6 Conclusion **95**
6.1 The road ahead 97

Appendices **99**

A Publications **101**

Bibliography **103**

List of acronyms

ABE attribute-based encryption

AE authenticated encryption

AES advanced encryption standard

AESM application enclave service manager

AMD advanced micro devices

ANOBE anonymous broadcast encryption

AOL America online

API application programming interface

ARM advanced RISC machine

ASLR address space layout randomisation

ASPE asymmetric scalar-product preserving encryption

ASID address space identifier

BE broadcast encryption

BIOS basic input/output system

CA certification authority

CDF cumulative distribution function

CBR content-based routing

CPU central processing unit

CSV comma separated value

CTR counter mode

DAG directed acyclic graph

DH Diffie-Hellman

DMA direct memory access

DoS denial of service

DRAM dynamic random-access memory

ecall enclave call

ECC elliptic curve cryptography

ECDH elliptic-curve Diffie–Hellman

ECDSA elliptic curve digital signature algorithm

EDL enclave definition language

EK endorsement key

EPC enclave page cache

ES encrypted state

FE functional encryption

GCM Galois counter mode

glibc GNU C library

GMP GNU multiple precision arithmetic library

GPG GNU privacy guard

GPU graphics processing unit

HE hybrid encryption

HE-PKI hybrid encryption with public key

HE-IBE hybrid encryption with identity-based encryption

HIBE hierarchical identity-based encryption

HMAC keyed-hash message authentication code

HTTP hypertext transfer protocol

IAS Intel attestation service

IND-CCA indistinguishability under non-adaptive and adaptive chosen ciphertext attack

IoT Internet of things

IBBE identity-based broadcast encryption

IBE identity-based encryption

I/O input and output

IP Internet protocol

IP intellectual property

ISA instruction set architecture

ISV independent software vendor

IV initialisation vector

JMH Java microbenchmark harness

JSON JavaScript object notation

JVM Java virtual machine

LDA latent Dirichlet allocation

LoC line of code

LLC last level cache

LTS long term support

MAC message authentication code

M&A merger and acquisition

MC memory controller

ME management engine

MEE memory encryption engine

MMU memory management unit

ocall outside call

OS operating system

PBC pairing-based cryptography

P2P peer-to-peer

PGP pretty good privacy

PIR private information retrieval

PKE public key encryption

PKI public key infrastructure

PMC performance monitoring counters

PRM process reserved memory

PSE platform services enclave

PSW platform software

pub/sub publish/subscribe

RAM random-access memory

REST representational state transfer

RMW read-modify-write

ROP return-oriented programming

RPC remote procedure call

RSA Rivest–Shamir–Adleman

RSS RDF Site Summary

RTT round-trip time

S3 simple storage service

SCBR secure content-based routing

SCONE secure container environment

SDK software development kit

SEV secure encrypted virtualisation

SGX software guard extensions

SHA secure hash algorithm

SME secure memory encryption

SoC system on chip

SRAM static random-access memory

SSD solid-state drive

SPI serial peripheral interface

STL standard template library

TA trusted authority

TCB trusted computing base

TCP transmission control protocol

TEE trusted execution environment

TLB translation lookaside buffer

TLS transport layer security

Tor the onion router

TPM trusted platform module

UUID universally unique identifier

VDR virtual data room

VM virtual machine

YCSB Yahoo! cloud serving benchmark

List of figures

2.1 TPM components. . . . 7
2.2 TrustZone scheme. . . . 8
2.3 AMD SEV architecture. . . . 9
2.4 SGX execution flow. . . . 10
2.5 SGX memory management. . . . 11
2.6 SGX paging. . . . 12
2.7 SGX caching effects. . . . 12
2.8 Overhead when starting SGX jobs. . . . 13
2.9 SGX local and remote attestations. . . . 14
2.10 SecureCloud platform architecture. . . . 17
2.11 Unlinkability: Tor. . . . 24
2.12 Indistinguishability: *(a)* TrackMeNot and *(b)* GooPIR. . . . 24
2.13 Quality of fake queries. . . . 25
2.14 Systems combining indistinguishability and unlinkability. . . . 26

3.1 SCBR overview. . . . 30
3.2 SCBR messages cycle. . . . 32
3.3 SCBR comparison: Native and SGX. . . . 33
3.4 Distinct workloads' characterisation. . . . 34
3.5 SCBR response to different workloads. . . . 34
3.6 Performance loss when surpassing the EPC limit. . . . 35
3.7 Enclave versus native running times for Lua benchmarks. . . . 37
3.8 Lightweight MapReduce components. . . . 38
3.9 MapReduce: Session establishment protocol. . . . 40
3.10 MapReduce: Provisioning of code and data. . . . 41
3.11 K-means with MapReduce. . . . 42
3.12 K-means example. . . . 43
3.13 MapReduce: Iteration duration. . . . 44
3.14 MapReduce: Cache misses. . . . 44
3.15 MapReduce: SGX overhead. . . . 45
3.16 Example of SecureStreams pipeline. . . . 46
3.17 Integration between Lua and Intel SGX. . . . 46
3.18 SecureStreams: Throughput comparison. . . . 48
3.19 SecureStreams: processing time. . . . 49

4.1 IBBE-SGX model diagram. . . . 54
4.2 IBBE-SGX: System components. . . . 55
4.3 IBBE-SGX initial setup. . . . 56
4.4 Comparison between distinct cryptographic protocols. . . . 57
4.5 IBBE-SGX: Partitioning mechanism. . . . 59
4.6 Performance of IBBE-SGX's bootstrap phase. . . . 61
4.7 Create and remove operations in IBBE-SGX. . . . 62
4.8 IBBE-SGX: Adding user to group and decryption. . . . 63
4.9 IBBE-SGX replay time for a given dataset. . . . 63
4.10 IBBE-SGX: Replay time of synthetic datasets. . . . 64
4.11 A-Sky architecture. . . . 66

4.12 A-Sky data model. 69
4.13 Throughput for administrative actions. 72
4.14 A-Sky enveloping throughput. 73
4.15 A-Sky: Throughput *vs.* latency. 73
4.16 A-Sky: Writing throughput to cloud storage. 74
4.17 A-Sky: User throughput. 74

5.1 X-Search architecture and execution flow. 79
5.2 Re-identification rates for X-Search and PEAS. 82
5.3 X-Search's accuracy. 82
5.4 Latency vs. throughput for X-Search and others. 83
5.5 X-Search memory usage. 83
5.6 Round-trip time for X-Search and others. 84
5.7 Cyclosa architecture and operating flow. 86
5.8 Privacy comparison between Cyclosa and others. 90
5.9 Accuracy comparison between Cyclosa and others. 90
5.10 Actual number of fake queries in Cyclosa. 91
5.11 Distribution of end-to-end delays for privacy-preserving systems. 92
5.12 Cyclosa: Impact of k on latency. 92
5.13 Throughput vs. latency of Cyclosa and X-Search. 93
5.14 Query protection vs. users blocked by search engine. 93

Chapter 1

Introduction

This work explores design strategies for distributed systems that leverage trusted execution environments (TEEs). We aim at achieving better security and privacy guarantees while maintaining or improving performance in comparison to existing equivalent approaches.

1.1 Context and motivation

Legacy systems perform access control by separating software roles into privileged and user modes. While privileged system software like hypervisors and operating systems (OSes) control machine resources, user-mode applications must rely on system software correctness, *i.e.*, that they are free of exploitable bugs, and honesty, *i.e.*, that they are not malicious.

The correctness aspect is already hard to verify. After all, system software is very complex and comprises millions of lines of code (LoC). To illustrate, more than 10 000 computer systems' vulnerabilities are being discovered each year [1] and the number of software bugs is proportional to its size. Presuming that an adversary would be able to exploit one of them is not a far-fetched assumption.

When it comes to honesty though, confidence may be even lower. With the raise of cloud computing, many computer systems are deployed on third-party infrastructure providers. The reasons for doing so are mostly related to the large costs of acquiring and maintaining private data centres. In such shared environments, at least the hypervisors [2] are under the provider's control. Even if one trusts the infrastructure provider and all of its employees with physical access or administrative credentials to the machines, still they would have to hope that other tenants running on the same platforms are not malicious.

A large code-base consisting of system software on which user applications must trust is therefore not the best approach from a security standpoint. TEEs, in turn, invert this logic. They make it possible for only a small piece of code to be considered safe, *i.e.*, belonging to the trusted computing base (TCB). This dramatically reduces the trust surface, as in this case it suffices to believe that the TEE hardware implementation is correct and has no backdoors, apart from having confidence on the reduced piece of software that runs in isolation, *i.e.*, within an *enclave*.

In the last quarter of 2015, a few months before this work started, Intel released the first commercially available machines with software guard extensions (SGX) [3]. It was the first TEE accessible in consumer-grade computers that offered out-of-the-box attestation mechanisms in conjunction with memory encryption, integrity and freshness guarantees. These assurances were attainable by user applications that could then be protected from higher privileged entities like the hypervisor or the operating system. Up to this date, it is still unmatched by alternatives (see Section 2.1).

With processes that coordinate over multiple interconnected machines, distributed systems face attacks on all fronts of their deployment. Reaching dependability [4] in such systems is not a trivial task. However, TEEs provide an opportunity to achieve this goal. This thesis explores these possibilities through the design and evaluation of experimental prototypes representative of fundamental distributed systems relying on TEEs—materialised with Intel SGX—to offer security and privacy. Doing so, we wish to answer the following questions:

- How can TEEs help to achieve security and privacy in distributed systems?
- What are the drawbacks?
- What benefits may come from using TEEs?

Assumptions

We assume that Intel SGX behaves correctly, *i.e.*, there are no bugs or backdoors. Additionally, we do not deal with side-channel attacks against SGX. We consider such attacks outside the scope of this dissertation and that the research community provides some solutions that could possibly be incorporated (see Section 2.2.6). Furthermore, we are also aware that denial of service (DoS) attacks cannot be prevented, *e.g.*, malicious agents might drop requests or refuse to initialise enclaves. Besides, we do not handle rollback attacks of persistent data. Standard solutions with monotonic counters could be used, although we are aware that they might be ineffective (see Section 2.2.5). Finally, we assume that all the cryptographic primitives and libraries used in enclaves are trusted and cannot be forged.

1.2 Contributions

The contributions of this work are as follows.

- Design, implementation and evaluation of **secure content-based routing (SCBR)**, the first system that demonstrates the practical benefits of SGX for privacy-preserving content-based routing (CBR). We protect compute-intensive matching operations in the trusted environment, so that efficient algorithms that operate on plaintext data can be used. As a consequence, we reach performance gains of one order of magnitude in comparison to a software-only solution with analogous guarantees. We also analyse SGX overheads when surpassing its memory limits. This work was done in collaboration with Christof Fetzer, from the Technical University of Dresden, Germany.
- Proposal of **Lightweight MapReduce**, a processing framework based on a programming model extensively used for parallel data processing in distributed environments. We ported a Lua interpreter engine to run inside secure enclaves and leverage it as execution unit that operates on code and data provisioned in encrypted form. From the usability perspective, a user can just write MapReduce scripts and let the framework handle data encryption and dissemination. Besides, we observe the performance influence of going beyond the last level cache (LLC) in enclave executions. Daniel Gravril and Emanuel Onica, from the Alexandru Ioan Cuza University of Iaşi, Romania, collaborated with this work.
- **SecureStreams**: a reactive middleware built on top of Lua libraries. Its architecture relies on Lua virtual machine (VM) pairs on each node, *i.e.*, one running inside enclaves and another outside. This way, only sensitive data processing is relayed to trusted environments, while message queuing and the pipeline management is kept outside. We analyse performance losses when compared to unsafe executions in terms of throughput and scalability. This was joint work with Aurélien Havet and Valerio Schiavoni, from our Institute, and Romain Rouvoy, from the University of Lille, France.
- Introduction of **IBBE-SGX**, a new cryptographic access control extension for collaborative editing of shared data. Thanks to TEEs, we are able to cut part of the computational complexity of an identity-based broadcast encryption (IBBE) scheme. Shielding a master key inside the trusted environment allows us to spare considerable computation time by avoiding the usage of an IBBE public-key during encryption. Because of this, we improve performance by orders of magnitude in comparison to hybrid encryption (HE), both in terms of membership changes and produced metadata, consequently also profiting in storage and network usage.
- To handle anonymity among group members, **A-Sky** is presented. Instead of relying on costly asymmetric cryptography like pretty good privacy (PGP), secure enclaves allow A-Sky to create key envelopes using efficient symmetric operations hence achieving faster execution times and shorter ciphertexts. In addition, we only require the usage of a TEE proxy for writing to the shared storage and leave the dominant data consumption operations directly in charge of rightful readers. We propose

Table 1.1: Papers related to this dissertation.

Chapter	Keyword	Section	Publication	Source code	Venue
2	SEV vs. SGX	2.1.3	[5]		SRDS'18
	Malware	2.2.6	[6]		SRDS'19
	SecureCloud	2.3	[7]		DATE'17
	Scheduling	2.3	[8]	[9]	ICDCS'18
3	SCBR	3.1.1	[10]	[11]	Middleware'16
	MapReduce	3.2.2	[12]	[13]	CCGRID'17
	SecureStreams	3.2.3	[14]	[15]	DEBS'17
4	IBBE-SGX	4.1.1	[16]	[17]	DSN'18
	A-Sky	4.2.1	[18]	[19]	SRDS'19
5	X-Search	5.1.1	[20]		Middleware'17
	Cyclosa	5.2.1	[21]		ICDCS'18

an end-to-end system based on micro-services and a representational state transfer (REST) interface before evaluating its performance and scalability. Both IBBE-SGX and A-Sky resulted from our collaboration with Stefan Contiu and Laurent Réveillère, from the University of Bordeaux, France. Additionally, Sébastien Vaucher, from our Institute, contributed in both.

- **X-Search**, in turn, leverages TEEs for providing a privacy-preserving solution for Web search. In order to prevent service providers from keeping accurate user profiles and therefore obstruct privacy breaches, we propose a SGX proxy between users and search engines. From the service provider's perspective, queries originate from another source, thus becoming more difficult to link them back to their issuing users. Since the proxy operates in the trusted environment, we can safely store past user queries and use them to obfuscate requests, so that the search engine cannot distinguish real from fake queries. These strategies combined offer stronger privacy guarantees and outperform previous approaches in latency and throughput. Our conjoint efforts with Sonia Ben Mokhtar and Antoine Boutet, from the University of Lyon, France, led to this work.

- Finally, we contribute with the proposal of **Cyclosa**. Instead of a centralised proxy, we now spread the load across a peer-to-peer (P2P) network of SGX relay nodes. Each one may issue their own queries through the decentralised network and also forward requests to the search engine on behalf of others, always having enclaves as intermediaries. Obfuscation is done through different paths, thus facilitating the delivery of results by simply discarding those that handle fake queries and therefore achieving perfect results' accuracy. Additionally, we propose an adaptive privacy protection solution based on sensitivity analysis that reduces the risk of user re-identification. With that, we solve the issue of possibly being blacklisted by search engines because of centralised proxies while meeting scalability and accuracy. In addition to Ben Mokhtar and Boutet, Sara Bouchenak—also from Lyon—contributed in Cyclosa. Likewise, David Goltzsche and Rüdiger Kapitza, from the Technical University of Braunschweig, Germany, joined the team. Valerio Schiavoni, from our Institute, contributed in both X-Search and Cyclosa.

These contributions were previously published in conference proceedings in papers co-authored by Pascal Felber and Marcelo Pasin, my supervisors, and me. Table 1.1 establishes the relation between publications and the corresponding section in this document where they are mentioned. Additionally, we indicate the publication venue and a reference to the corresponding source code, when available.

1.3 Outline

This manuscript is organised in six chapters, the first being this introduction. The remaining chapters are arranged as follows.

Chapter 2 presents the background on TEEs and particularly on Intel SGX. We focus on their main features and peculiarities, which finally endorse our choice for SGX. The SecureCloud project is briefly described before we present the state-of-the-art by mentioning related work specific to each of our contributions.

In **Chapter 3**, we explore the cloud scenario depicted in Section 1.1 by designing and evaluating systems supposed to be deployed in such unreliable environments. A communication middleware (Section 3.1) and two distributed processing frameworks are presented (Section 3.2). Section 3.1.1 presents SCBR, where we essentially put a publish/subscribe (pub/sub) matcher unit inside an enclave, *i.e.*, the component that handles sensitive data. The processing frameworks, one for batch execution (Section 3.2.2) and another for processing stream of events (Section 3.2.3), used a Lua script interpreter within enclaves, so that code and data can come on demand. The goal in this chapter is to quantify the SGX performance implications in the context of practical systems, besides learning about the main design concerns involved in building these secure applications. Our results show that whenever cache and enclave page cache (EPC) limits are surpassed, there are performance penalties. Although not surprising, these results—along with the designs—contribute towards sensitive data processing in untrusted public clouds.

TEEs are not only useful for shielding legacy artefacts like script processors. In fact, simply porting memory-eager systems may not be a good idea as they suffer considerable overheads in the current version of SGX. Instead, in **Chapter 4**, we capitalise on a trusted environment for protecting cryptographic keys. Particularly, for group communication and data sharing. In Section 4.1.1, we generate a master secret within an enclave, from where it never leaves in unencrypted form. Since this key can only be used in protected execution, we can cut some computational cost of an IBBE scheme by using simpler cryptographic primitives. That brings significant advantages when compared to HE in terms of metadata size and performance. At the same time, it obviates the need for relying on a public key infrastructure (PKI). Apart from that, we also design in Section 4.2.1 a distributed system that offers anonymity among group members. For that, we use enclaves to harbour keys and group membership data while attaching some metadata to encrypted files that can then be stored in untrusted clouds. A user who is member of the group for a given file will be able to retrieve the key from the attached metadata. The take-away message is that TEEs, apart from shielding traditional applications, can also have an essential role in the design of efficient cryptographic protocols.

Until this point, we covered the design of distinct server-side systems. Nevertheless, often times users must rely on established services which are unlikely to change. That is the case for web search engines, which are able to quietly track user activities in spite of correctly answering their queries. Because of this, **Chapter 5** handles privacy-preserving solutions from the user perspective. Our first approach, described in Section 5.1.1, adds a centralised proxy to hide user identities from search engines. At the same time, we obfuscate requests using past queries that are securely kept within enclaves, so that the introduced noise hinders the service provider's ability to keep meaningful profiles. But a toll must be paid: the introduced fake queries have consequences in the quality of results, which need to be filtered. Moreover, its centralised nature allows search engines to easily block the SGX proxies, as the proxies potentially issue considerable amounts of traffic. Besides, with a growing number of users they would become performance bottlenecks. To counter these disadvantages, we propose in Section 5.2.1 a fully decentralised approach. It consists in a user-side peer-to-peer solution that combines performance, scalability, fault tolerance and accuracy of results, since in this case no result filtering is needed.

Finally, **Chapter 6** revisits our achievements on the three parts: cloud deployment systems, crypto schemes and client-side privacy protection. Before concluding, we discuss some avenues for further research.

Chapter 2

Background and related work

Integrity and confidentiality of applications are enforced by means of logical isolation. Virtual address spaces and privileged instructions, for instance, are hardware mechanisms used by operating systems (OSes) to prevent unauthorized processes from getting access to potentially sensitive pieces of memory (*e.g.*, belonging to other users) or operations (*e.g.*, interacting with input and output peripherals). Likewise, hardware virtualisation extensions [22, 23] are used by hypervisors to isolate multiple OSes running on the same physical host. Such a model requires that both the OS and the hypervisor be trusted by a user application, since it uses their services.

In multi-tenant setups, as in cloud environments, at least the hypervisor is under the provider's control. Despite the legal protection assured by possible contractual agreements between applications and infrastructure providers, security concerns may arise from this separation of administrative domains. The potential threats are numerous: distinct law jurisdictions, corrupt employees with privileged access, malicious co-located tenants etc.

From a technical perspective, OSes and hypervisors are composed of millions of lines of source code, resulting in bloated trusted computing bases (TCBs). The number of bugs, which may be reduced by means of OS formal proofs [24], is proportional to software size [25]. Other tenants running on the same computers can profit from them to gain access to sensitive data. Moreover, system administrators of the cloud provider have access to all application data. Hosted applications can further be compromised either by privileged administrators acting maliciously or as a consequence of having their credentials hijacked [26].

2.1 Trusted execution environments

Traditional hardware isolation mechanisms protect user applications from one another (memory control), multiple OSes from one another (virtualisation), and the system software from user applications (privileged instructions). However, none of them provide isolation to user applications from the system software. Apart from logical isolation, other mechanisms must also be employed, for instance, when attackers get physical access to platforms. In such cases, they could wiretap memory buses or read memory content through cold boot attacks [27, 28].

Cryptography is widely used to protect data in transit, *e.g.*, using transport layer security (TLS) [29], or at rest, for storage. When data are processed though, ciphertexts must be deciphered before loaded into system memory. During processing, data confidentiality is hence threatened by privileged users and physical attackers. Cryptographic approaches like homomorphic encryption could be employed, although they are not considered practical due to their prohibitive cost [5].

To solve issues like isolation of user applications from system software and providing cryptographic layers of isolation against physical attacks, trusted execution environments (TEEs) were proposed. Different implementations vary in terms of features, which we summarize in Table 2.1. In this overview, we leave aside academic proposals (*e.g.*, Sanctum [30], Bastion [31] and AEGIS [32]) and focus on popular commercial solutions:

Table 2.1: Comparison of TEEs.

	TPM	TrustZone	AMD SEV	Intel SGX
Released	2009	2005	2016	2015
Running mode	coprocessor	special mode	hypervisor	user-level
ISA	n/a	ARM	x86_64	x86_64
Executes arbitrary code	✗	✓	✓	✓
Secret hardware key	✓	✗*	✓	✓
Attestation and Sealing	✓	✗*	✓	✓
Memory encryption	n/a	✗	✓	✓
Memory integrity	n/a	✗	✗	✓
Resilient to wiretap	✗	✗	✓	✓
I/O from TEE	✓	✓	✓	✗
TEE usable memory limit	n/a	system RAM	system RAM	93.5 MiB
TCB	Data and funcs in the TPM chip	OS and apps in the secure world	Entire VMs	Trusted app partition

n/a: not applicable

*: possible with additional hardware

- Trusted platform modules (TPMs) are dedicated coprocessors, and therefore do not allow for the execution of general purpose software. Their TCB boundary is the microcontroller package, which contain the memories they need to operate. Memory integrity and confidentiality guarantees serve hence no purpose, as memories are inside this package and cannot be tampered with or wiretapped without physical violation. Wiretapping its input and output (I/O) bus, however, could reveal sensitive information, *e.g.*, data unsealed by the TPM. Besides, the TPM state can only be reset by physical presence, which makes it suitable for preventing remote attacks. The TPM specification was standardised in 2009 through ISO/IEC 11889.

- TrustZone is a TEE for the advanced RISC machine (ARM) instruction set architecture (ISA) that switches between secure and unsecure modes. It provides isolation by means of interrupts routing and restrictions on the memory bus and memory management unit (MMU). However, there is no built-in cryptographic primitives that could provide a root of trust for persistent sealing and attestation [33]. Since ARM is a intellectual property (IP) core provider and not a chip manufacturer, vendors might implement such additional security, although most often they are not publicly disclosed. In 2018, ARM announced families of semiconductor IPs called CryptoIsland and CryptoCell that can possibly be integrated to ARM central processing units (CPUs) in a single system on chip (SoC). Conceptually, that would be like having a TPM inside the processor package, allowing the usage of hardware keys as root of trust (like Intel software guard extensions (SGX)).

- AMD secure encrypted virtualisation (SEV) provides automatic inline encryption and decryption of memory traffic, granting confidentiality for data in use by virtual machines (VMs). Cryptographic operations are performed by hardware and are transparent to applications, which do not need to be modified. Keys are generated at boot time and secured in a coprocessor integrated to the SoC. It was conceived for cloud scenarios, where guest VMs might not trust the hypervisor. Apart from including the whole guest OS in the TCB, it does not provide memory integrity and freshness guarantees. Rowhammer attacks [34] might corrupt data and rollback attacks are not detected.

- Intel SGX is primarily conceived for shielding micro-services, so that the TCB would be minimised. Automatic memory encryption and integrity protection are performed by hardware over a reserved memory area fixed at booting time, defined in the basic input/output system (BIOS) and limited to 128 MiB. Whatever is kept in this area is automatically encrypted and integrity checked by hardware. The trust boundary is the CPU package, which holds hardware keys upon which attestation and sealing services are built. Applications are partitioned into trusted (shielded in *enclaves*) and untrusted parts. The OS is considered untrusted, which prevents enclaves from directly issuing system calls. SGX enclaves are subject to side-channel attacks and denial of service.

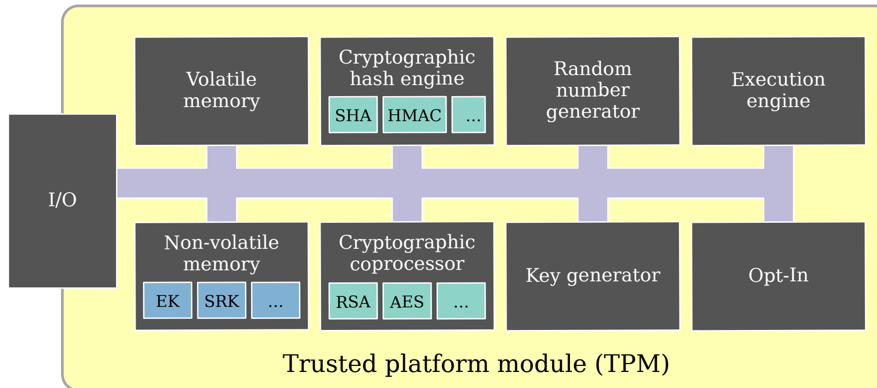

Figure 2.1: TPM components.

We further detail these technologies in the following sections.

2.1.1 Trusted platform modules

A TPM [35] is an independent and specialized tamper-resistant coprocessor. Its root of trust comes from an integrated asymmetric pair of endorsement keys (EKs) physically burnt in the component. The private endorsement key is never made available to users, irrespective of their system privileges. Rather than providing general purpose computations, TPMs perform a small set of security operations such as random number generation, cryptographic hash functions, *e.g.*, secure hash algorithm (SHA), public- and symmetric-key cryptographic algorithms, *e.g.*, advanced encryption standard (AES) and Rivest–Shamir–Adleman (RSA). Some implementations also provide trusted time and monotonic counters (see Section 2.2.5). Typically, a TPM securely holds software measurements and cryptographic keys which may be used for trusted boot, remote attestation and data sealing.

Access to TPMs is based on the *ownership* granted to whom first sets a shared secret. Access control is operated by a component called *Opt-In*, which maintains flags associated to the TPM's state. When someone takes ownership, a storage root key (SRK) is created and can be used for sealing data in persistent storage. Only the owner is able to remotely use the TPM in its full capabilities, although it may be factory reset through the assertion of physical presence. Nevertheless, other users can perform operations allowed by the owner, like querying software measurements, storing keys and using crypto primitives. Figure 2.1 illustrates internal components of a TPM.

Being a separate component, TPMs are detached from other isolation mechanisms such as memory page tables and privileged instructions. Security is therefore highly dependant on how TPM chips are wired to main processors, what are the privileges users must hold to get access to such chips and whether adversaries might have physical access. Wiretapping the TPM's I/O lines may compromise sensitive data.

2.1.2 ARM TrustZone

ARM TrustZone [36, 37] is a popular TEE, mostly used in low-energy or mobile devices. It provides a protection domain called *secure world* that cannot be accessed by the *normal world*. Each physical processor core is viewed as two virtual ones: secure and non-secure, which are used in a time-sliced manner. The context switch is made through a *monitor mode* (ARMv8-A) accessed by a special instruction, or through hardware exception mechanisms (ARMv8-M). At booting time, a secure firmware initialises the platform and decides what is part of secure and non-secure worlds by configuring the interrupt controller and setting up memory partitions and peripherals. Then, the processor switches to the normal world, typically yielding control to the normal (or rich) OS bootloader.

In the system bus, a supplementary control signal (a 33rd bit) called Non-Secure bit (NS) is set by hardware whenever a transaction (read or write) is made by components that belong to the normal world. Being so, once addresses are decoded, they cannot match any component that is part of the secure world. Additional mechanisms provide the means for securing memory regions and interrupts, which are also partitioned

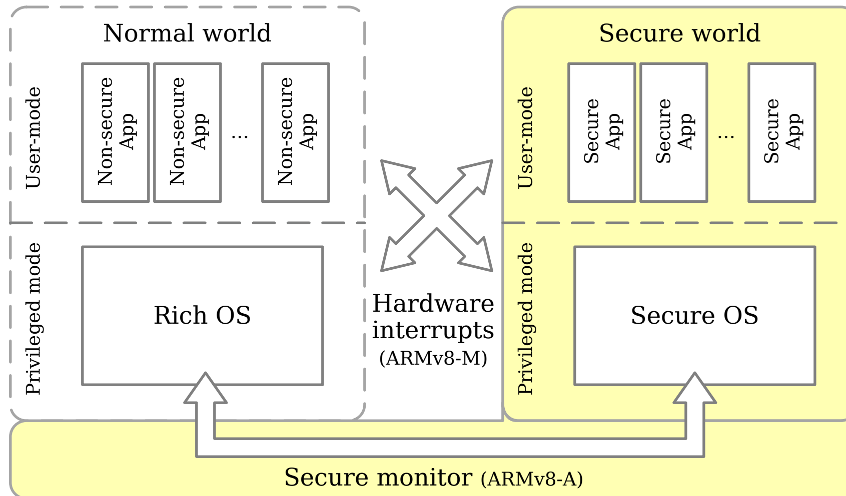

Figure 2.2: TrustZone scheme.

between the two worlds and can only be configured in secure mode. Such mechanisms provide security building blocks such as trusted I/O paths and secure storage.

Remote attestation and sealing do not come out of the box, although some works simply assume the availability of a secure hardware key in the secure world [38, 39], since ARM chip vendors are free to add their own IPs to manufactured SoCs. In conjunction with a TPM that can solely communicate with the secure world, one can envision a root of trust for providing these services [40].

TrustZone provides, however, only one secure world. As a consequence, multiple secure applications need to share it, increasing the chances of dividing the protected environment with potential attackers. Alternatively, a requirement of at most one application in the secure world can be enforced, although this could be too restrictive in some scenarios. Moreover, the secure world is under the control of a separate OS, which means that it can still be accessed by some (possibly compromised) system administrator.

2.1.3 AMD SEV

The advanced micro devices (AMD) SEV protects data in use by VMs. It relies on a hardware encryption engine embedded in the memory controller (MC) which automatically performs cryptographic operations, given that appropriate keys are provided to it, and adds minimum performance impact and no requirement for application changes. Key generation and management are performed in the AMD secure processor, a dedicated security subsystem based on an ARM Cortex-A5 coprocessor physically isolated from the rest of the SoC. It contains a dedicated random-access memory (RAM), non-volatile storage in a serial peripheral interface (SPI) flash and cryptographic engines. Figure 2.3 depicts the high level architecture.

While the secure memory encryption (SME) feature uses a single key for system-wide memory encryption, SEV involves multiple encryption keys, one per VM. When using SME, ephemeral keys are randomly generated at boot time by the secure processor and are used to encrypt memory pages that are marked by system software within page tables through the *C-bit*. Alternatively, all pages may be encrypted if the Transparent SME, or TSME, is activated during boot time. TSME does not require modifications in the OS and prevents physical attacks like cold boot.

Unlike SME, SEV allows the association of one encryption key per hardware virtual machine in a way that the hypervisor has no longer access to everything within guest VMs. The same cryptographic isolation is present in the other direction, *i.e.*, the hypervisor has a separate key that prevents the guest from reading its assigned memory pages even in case of a malicious attack resulting in the break of memory logical isolation. Despite that, guests and the hypervisor still communicate, *i.e.*, the latter performs scheduling and device emulation for VMs.

Apart from resources management, the hypervisor also manages keys, although it has no access to them. This is done by the manipulation of address space identifiers (ASIDs), which are specified when VMs are

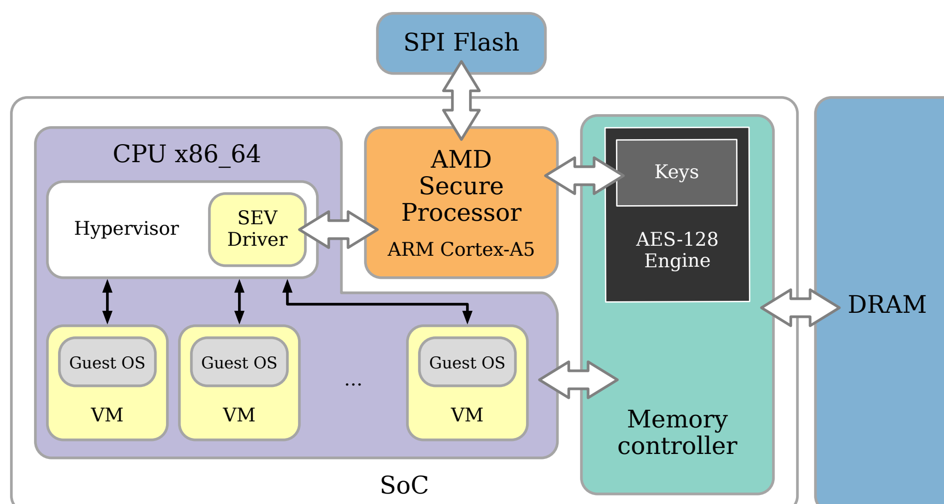

Figure 2.3: AMD SEV architecture.

launched and are used to determine the encryption key. Essentially, ASIDs are used as indexes into a key table in the MC that determines the key in use by the AES engine. Besides, such identifier is also employed to tag data in the cache, in a way that only their owner is able to hit on that cache line and get access to the corresponding data.

Memory pages containing instructions and guest page tables are always private, so that attacks such as code injection become more difficult. Data pages, on the other hand, may be shared by each guest OS through the C-bit manipulation. In particular, this is used for direct memory access (DMA), which is only allowed in shared memory pages. From the software perspective, this requires changes in the guest OS and hypervisor, so that they can set memory pages access privileges. That is done through an open source AMD secure processor driver, which communicates with the secure processor firmware. The AMD secure processor only runs signed firmware, which is a closed source package that AMD provides to BIOS vendors.

Key management allows for platform authentication, virtual machines start-up, migration and snapshot. Chips are fused with unique keys, including the chip endorsement key (CEK) derived from chip-unique values and an AMD signing key for authentication. Apart from that, platform owners may set a certification authority (CA) key which may be used for allowing migration exclusively to other platforms that had been signed by the same CA. Once combined, the CEK and the CA key produce the platform endorsement key (PEK) which is encrypted and stored in the SPI flash. The PEK is finally used to derive the platform Diffie-Hellman key (PDH) in conjunction with guest OSes every time the machine is powered on or reset. That way, the hypervisor is not able to intercept communication between guest OSes and the AMD secure processor.

Originally, the processor register state of VMs could be exploited by hypervisors running under AMD SEV. For that reason, AMD released the SEV encrypted state (ES) that allows guests to restrict access to the register state, albeit possibly sharing it with hypervisors. Although AMD SME/SEV/SEV-ES allow memory and registers encryption and platform attestation to a finer granularity (VMs), these technologies are vulnerable to memory integrity manipulation (*e.g.*, rowhammer) and rollback attacks, *i.e.*, when a previous state (even if encrypted and signed) is replayed back and considered genuine.

2.2 Intel SGX

Intel SGX provides a TEE by employing distinguished approaches, particularly with respect to running privileges (user-level) and security guarantees (memory integrity and freshness). Limiting the secure execution to user-level processes allows to purge the OS out of the TCB, therefore dramatically reducing the size of the latter. Memory integrity and freshness in conjunction with confidentiality guarantees, on the other hand, make the secure environment robust against all kinds of memory tampering. These features,

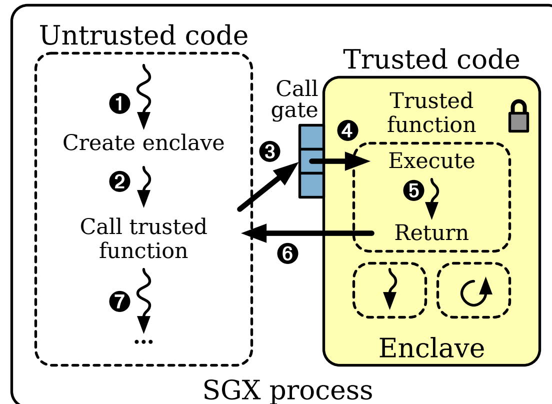

Figure 2.4: SGX execution flow.

however, are not costless. A unit of code protected by SGX, or *enclave*, often needs to use OS services, and such interactions become more expensive. Likewise, the bookkeeping that enables memory safety is made by hardware, hence consuming silicon area and memory at runtime. Because of that, enclaves are limited to use a small amount of protected memory. Whenever they need to use more than that, considerable overheads arise.

SGX aims to shield code execution against inspection and tampering from privileged code (*e.g.*, infected OS) and certain physical attacks. It does so by offering an instruction set extension in x86_64 CPUs manufactured by Intel. SGX seamlessly encrypts and tracks integrity tags of memory in use by enclaves. In this way, it is guaranteed that at any time during enclave execution the most recent and untampered plaintext data is present inside the CPU package. As a consequence, computations done in the enclave cannot be seen from the outside and any modification attempts are detected, including replay of previously authenticated values. The TCB of an SGX enclave is composed of the CPU itself, and the code running within. The assumption is that opening the CPU package is difficult for an attacker, and leaves clear evidence of the breach.

To enable an application to use enclaves, the developer must provide a signed shared library (`.so` or `.dll`) that will execute inside an enclave. The library itself is not encrypted and can be inspected before being started. Due to this, no secret should be stored inside the code. An enclave is provided with secrets, like certificates and keys, with the help of a remote attestation protocol. This protocol can prove that an enclave runs on a genuine Intel processor with SGX and can verify that its identity matches that of the code that is expected to run [41]. As a result of the attestation, a shared secret that enables a secure communication channel is established and permits the remote entity to provide the enclave with encrypted secrets.

SGX applications are partitioned between trusted and untrusted segments. Typically, the trusted part is supposed to handle sensitive data and computations, whereas the untrusted is responsible for performing system calls and handling non-sensitive backing operations. While trusted code has access to the whole process memory space, the untrusted code is limited to its own memory pages. Transferring the control from one running mode to the other happens in a similar fashion to remote procedure calls (RPCs) in the sense that arguments are serialised and replicated in another memory space before switching modes. Calls made from the untrusted code to the trusted one are called enclave calls (ecalls), whereas the ones in the opposite direction are called outside calls (ocalls). Figure 2.4 depicts the basic execution flow of SGX. First, an enclave is created (❶). As soon as a program needs to execute a trusted function (❷), it performs an ecall (❸). The call goes through the SGX call gate to bring the execution flow inside the enclave (❹). Once the trusted function is executed by one of the enclave's threads (❺), its result is encrypted and sent back (❻) before giving the control to the main processing thread (❼).

Intel provides a set of tools to aid the coding and fulfilment of SGX requirements. The SGX software development kit (SDK) [42] comprises a generator for proxies and stubs written in C that are supposed to be compiled and linked to both trusted and untrusted code. This generation is based on a text-based configuration file that follows the syntax of the enclave definition language (EDL), which basically defines

the interface of edge routines. Since system calls and I/O instructions are not allowed inside enclaves, Intel also provides libraries that are guaranteed to comply with these limitations. It also contains wrappers for specific functions, like accessing the true random number generator by the `RDRAND` instruction, which uses hardware noise as source of entropy. Moreover, it provides functionalities to improve performance like the switchless calls [43], which basically implement asynchronous handling of ocalls to reduce the number of transitions between trusted and untrusted modes. The SDK also includes the *enclave signing tool* responsible for measuring and signing the shared library to be loaded as an enclave. In the following sections, we further detail some of the SGX features, their implications and limitations.

2.2.1 Memory protection

Encrypted memory is provided in a reserved memory area predefined at boot time. It is called process reserved memory (PRM) and is limited to 128 MiB in the first version of SGX, although future releases might relax this limitation [44]. Within the PRM, an area of at most 93.5 MiB [8] called enclave page cache (EPC) can be used by application's memory pages, while the remaining area is used to maintain SGX metadata. If this limit is surpassed, enclave pages are subject to a swapping mechanism implemented in the Intel SGX driver, resulting in severe performance penalties [10, 45, 46].

Normally, memory transactions that miss the cache are handled by the processor's MC. If they correspond to PRM addresses though, a component of the MC called memory encryption engine (MEE) takes over [41]. Figure 2.5 illustrates this mechanism. The MEE is responsible for providing cryptographic operations, tamper-resistance and replay protection. It generates random keys at every boot, one for cryptographic operations and another for message authentication codes (MACs). The safety guarantees are achieved by (i) encrypting data before sending them to dynamic random-access memory (DRAM); (ii) updating an integrity tree; (iii) decrypting data fetched from DRAM; and (iv) verifying the integrity tree.

An integrity tree is typically implemented as a Merkle tree [47], where each node holds a hash digest of its children. The root is the digest of the entire data, and each leaf the hash of one data unit. Being so, an update requires to re-compute only the implicated leaf and its ancestors. Instead of simple hashes, SGX MEE uses stateful MACs. The state is a *nonce* (non repeating number, coined to be used only once) called by Intel as the *version* of each data unit, which has the size of a cache line, *i.e.*, 64 B. The tree is stored in untrusted memory, except for its root that is kept in an internal on-die static random-access memory (SRAM) and inaccessible from outside. It reflects the integrity of the most recently written protected area at any given time. Any mismatch during a verification causes a MC lock, which ultimately requires a machine reboot, so that the MEE restarts with fresh keys [41].

Accesses to enclave pages that do not reside in the EPC trigger page faults. The SGX driver interacts with the CPU to choose which pages to evict. Such pages of 4 KiB are moved to main memory through the privileged instruction `EWB`, which also computes a cryptographic MAC of the page and stores it in

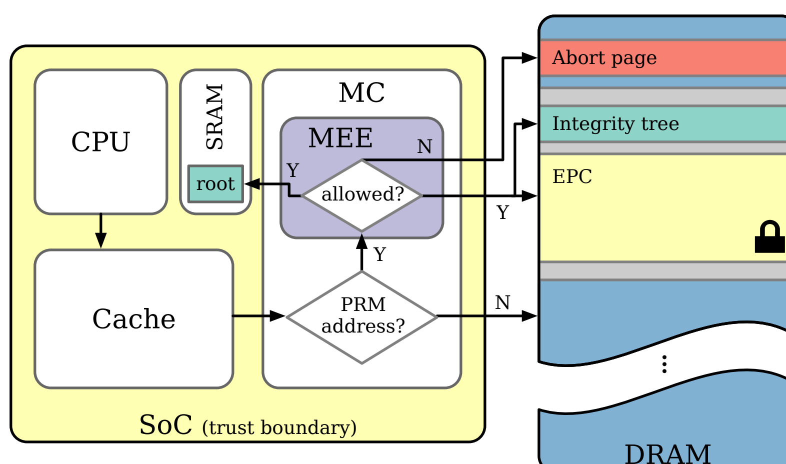

Figure 2.5: SGX memory management.

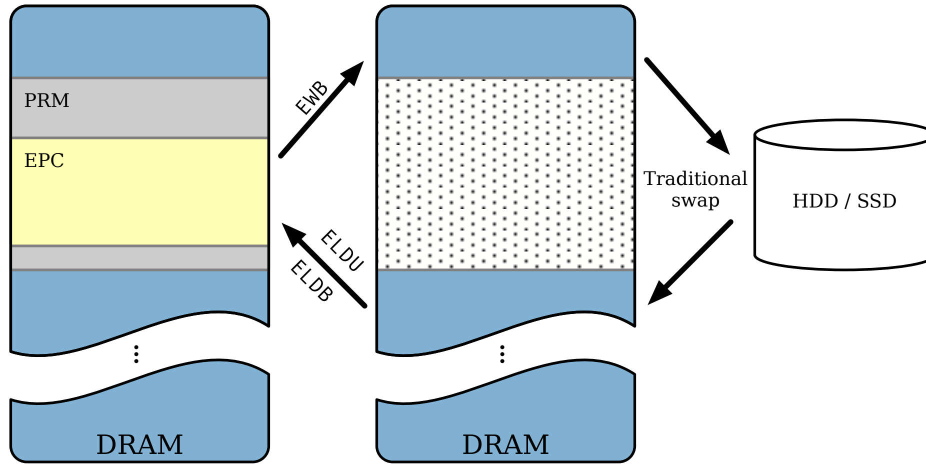

Figure 2.6: SGX paging.

unprotected memory. Additionally, EWB generates a nonce of 8 B, or *page version*, that ensures the freshness of the evicted page once it is loaded back in. These nonces are stored in special EPC pages called *Version Arrays*, which are not assigned to any enclave and therefore cannot be read by software. After a page is evicted, the SGX driver loads the requested page, *i.e.*, the one that triggered the fault, from main memory through the instructions ELDB or ELDU. These, in turn, decrypt the page and perform integrity and freshness checks. Figure 2.6 illustrates memory swaps.

Despite the close interaction between the SGX driver and the CPU for paging, one does not need to trust the driver since it cannot violate the confidentiality nor the integrity of enclave pages, which are maintained and checked by hardware. External snooping, such as eavesdropping the bus or the EPC memory content, will only reveal ciphertext undistinguishable from random data. Likewise, data modifications or feeding integral old data will be detected through mismatching authentication codes.

2.2.2 Limitations and performance implications

The main performance bottlenecks when using Intel SGX appear during transitions between trusted and untrusted modes (inside/outside the enclaves) and under intensive memory usage, notably in two stages: (i) when exceeding the processor's last level cache (LLC), which requires cache evictions and DRAM fetches, along with cryptographic and integrity operations; and (ii) when exceeding the EPC size, which triggers memory swaps serviced by the underlying OS.

Transitions are essential as enclaves cannot perform system calls. This is one of the main design concerns when porting legacy applications to SGX enclaves. Every disk or network access, for instance, must be handled by untrusted code. While switches between user and privileged modes when executing system

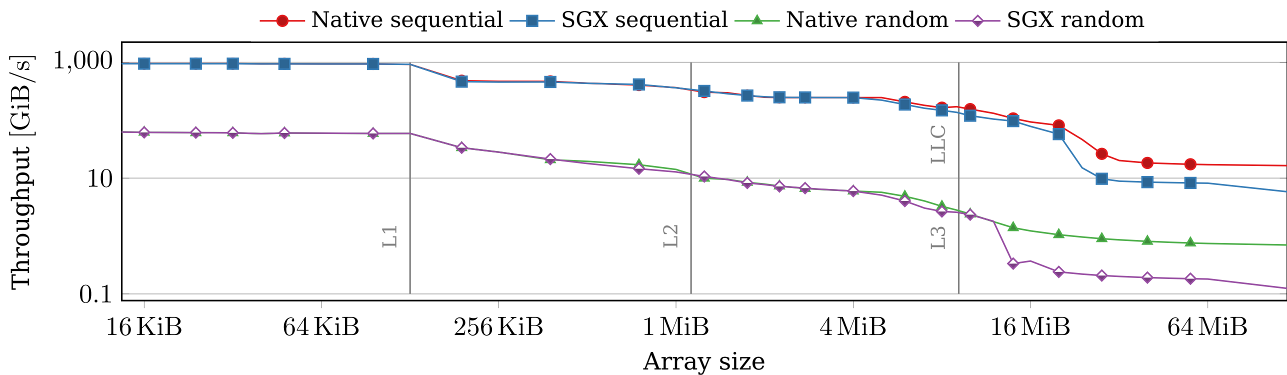

Figure 2.7: SGX caching effects according to sequential and random reads.

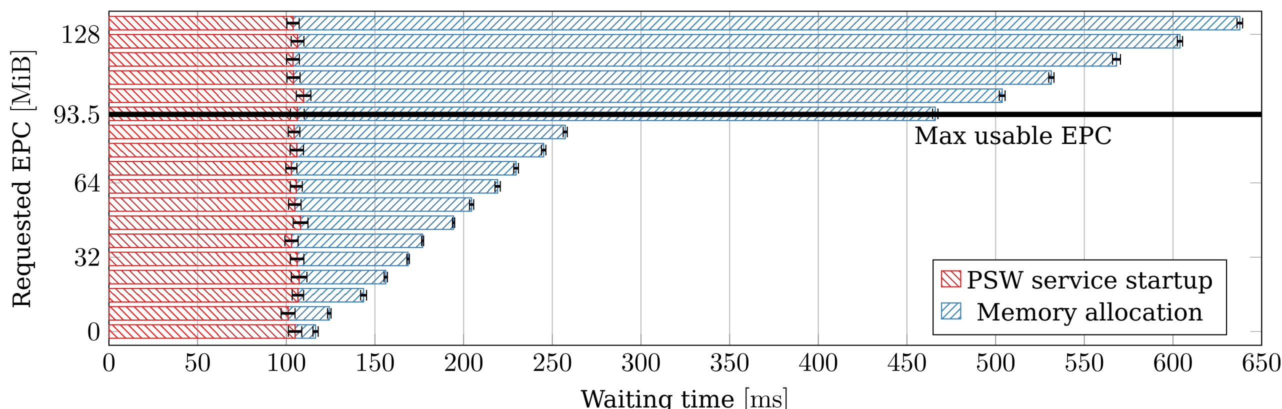

Figure 2.8: Overhead when starting SGX jobs.

calls are completed in 150 CPU cycles [48, 49], enclave transitions (`EENTER`, `ERESUME` or `EEXIT` instructions) take more than 3000 cycles each [50], *i.e.*, an ocall, where the control leaves and enters back in the enclave, is more than $40\times$ slower in comparison to system call transitions. This does not take into account the extra work performed in edge routines (*e.g.*, checking memory bounds) and collateral costs of enclave transitions like the ones caused by flushing the translation lookaside buffer (TLB) for security reasons.

To mitigate this issue, several systems [46, 50, 49] proposed the asynchronous treatment of transitions, which was later integrated to the Intel SGX SDK 2.2 as a feature named *switchless calls* [43]. This is essentially done by (i) having at least one thread inside and another outside the enclave; (ii) establishing a communication channel between them (shared memory in untrusted area); and (iii) using synchronisation primitives to signal events (spin locks). If we take an ocall as an example, an enclave thread would enqueue a request in a data structure residing in untrusted memory. The request is then acquired and processed by an untrusted worker thread which puts the result back in the shared memory. At this point, an enclave thread can collect and consume the result. This approach, although simple, brings along several practical matters.

While considerable performance improvements of using switchless calls can be observed in heavy workloads, gains are diminished when asynchronous calls are sparse. In idle workloads, a lot of time and energy is spent in busy waiting. These CPU cycles could instead be used by other threads or processes to perform more useful tasks. Moreover, as the state of enclave threads are kept in protected memory, the amount of active threads is limited. Besides, the optimal number of worker threads also depends on workloads, which are not always easy to estimate. For these reasons, Intel SGX SDK leaves for the user to decide whether edge routines are switchless by marking them with the keyword `transition_using_threads` in EDL files.

Regarding memory usage, we performed an experiment to observe caching effects of employing enclaves [5]. It consists of reading through a fixed amount of memory in sequential and random access patterns within an enclave and natively. Both runs were done in the same machine, an Intel Xeon E3-1275 v6, with 16 GiB of RAM. Figure 2.7 shows the observed throughput averaged over 10 runs. Within the cache, performance in the trusted environment is strictly equivalent to native, particularly up to the L2 cache limit. When the amount of memory surpasses the LLC, throughput is greatly affected, with random accesses incurring in bigger overheads than sequential reads.

In the first version of SGX, the whole memory reserved to an enclave must be allocated at the initialisation of the trusted environment. This constraint imposes some issues, as some services react to instant loads that vary over time. Having to set memory allocation beforehand may lead to exhaustion in case of underestimation or to underused pages otherwise. Besides, it also brings some start-up waiting time that would be diluted across the process lifetime in case it supported dynamic allocation. This limitation, along with the impossibility of changing page permissions (read, write and execute), guarantees that the initial state that is certified by attestation (see Section 2.2.3) remains unchanged throughout the enclave's execution.

SGX2 extensions [44], on the other hand, support dynamic allocation. To this date, we had no access to SGX2 hardware.

To quantify the described initial overhead, we conducted an experiment [8]. Apart from memory allocation, we also consider the support service initialisation time. Intel SGX SDK provides the platform software (PSW) that includes the application enclave service manager (AESM), which needs to be bootstrapped before enclaves are deployed. It assists SGX processes in enclave initialisation, attestation and supporting the access to platform services (see Section 2.2.5). Figure 2.8 shows the average results for 60 runs of docker containers that first start AESM and launch enclaves that allocate different amounts of memory. Error bars represent the $95\,\%$ confidence interval. As expected, the service startup time is virtually the same in all runs, accounting for about $100\,\mathrm{ms}$. Memory allocation time, on the other hand, shows two clear linear trends: before and after reaching the usable EPC memory limit. Until this limit, the time increase rate is $1.6\,\mathrm{ms/MiB}$ after which it jumps to $4.5\,\mathrm{ms/MiB}$, plus a fixed delay of about $200\,\mathrm{ms}$. Note that these times are just for *allocating* memory, before any real use. Higher overheads are expected when memory pages need to be swapped (see Section 2.2.1), and we evaluate this behaviour in a full-fledged publish/subscribe (pub/sub) system in Section 3.1.1.

2.2.3 Enclave signing and attestation

The development of applications targeted to run within SGX enclaves, besides the usual iterations on coding and compiling, also includes a mandatory signing step before the executables are able to be deployed and used in production. This essentially serves two purposes: *(i)* the code is uniquely associated to an independent software vendor (ISV), making it recognisable by customers and accountable for any consequence originated from its product; and *(ii)* whoever communicates with the application can have guarantees that the endpoint inside the enclave has loaded and is actually running the expected code within a genuine SGX platform.

The signing material includes information about the vendor, the date, some attributes, a version number and the enclave measurement, which corresponds to a digest (SHA-256) made upon the enclave's initial state, including data, code and metadata (security flags associated with memory pages) [51, 52]. When the enclave is loaded, a hardware implementation of the same measurement operates on the actual content of the running enclave (`MRENCLAVE` register), which has to precisely match the one that was computed during the signing step. Upon measurements matching, a hash of the signer's public key is computed and stored in the `MRSIGNER` register. These two registers reflect the enclave code and its author.

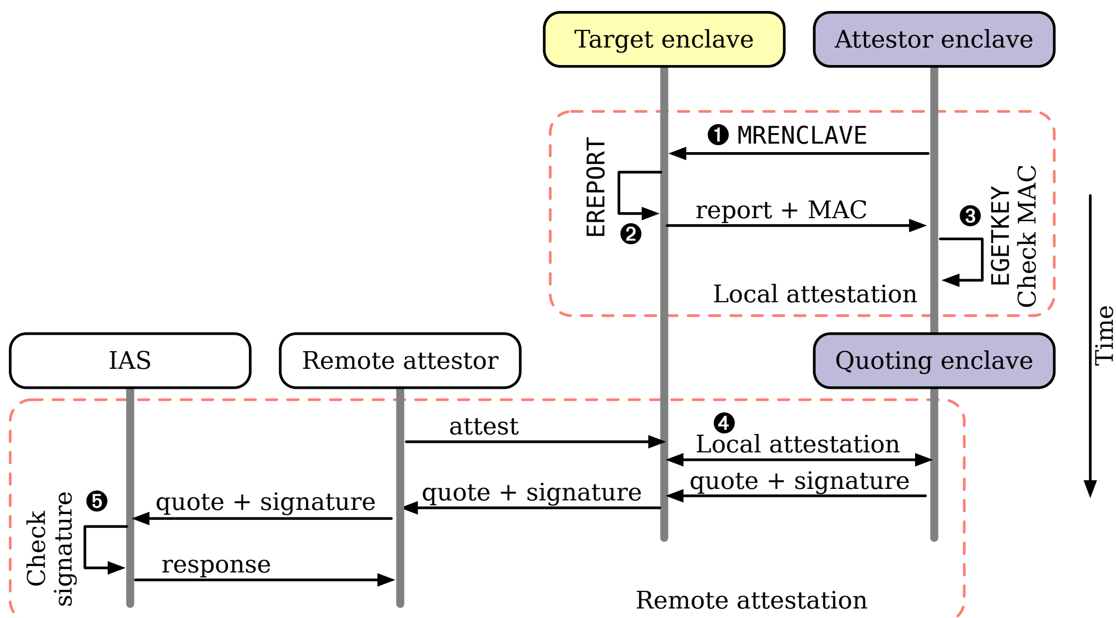

Figure 2.9: SGX local and remote attestations.

Later on, when interlocutors want to communicate with a running enclave, they should first attest it before sharing sensitive data with it. Attestation is therefore an essential requirement for virtually any enclave deployment. It can be performed locally or remotely, the latter being dependent on the former (Figure 2.9). The local procedure is based on a symmetric key and starts when the attestor sends its identity (❶) to the enclave being attested (target). This, in turn, calls the instruction `EREPORT` (❷), that cryptographically binds its identity (`MRENCLAVE` and `MRSIGNER`) and other attributes in a MAC tag that can only be locally checked by the attestor. This guarantee comes from the fact that the key used for computing the MAC can only be obtained by the `EGETKEY` (❸) instruction issued by the attestor on the same SGX platform. Such instruction retrieves keys that derive from the processor specific key hierarchy and can only be executed inside an enclave.

In case of remote attestation, asymmetric keys are used. A *quoting enclave* (installed along with the PSW) first performs a local attestation (❹) and creates another report called a *quote*. The quote, in turn, may be checked by the remote party with the aid of an online service called Intel attestation service (IAS), which checks the signature affixed to the quote (❺). The protocol also allows the enclave being attested to append arbitrary additional data to the quote. As the establishment of a secure channel is often desirable once the trust is settled, a sensible approach would consist of the following steps: (i) target enclave generates a pair of asymmetric keys; (ii) the public part of such key is sent along with the quote; (iii) once the attestation is successful, the key from the quote is used by the attestor to provide the enclave with any secrets, possibly a symmetric communication key. As the corresponding private key is securely shielded within the enclave, no third party would be able to inspect further interactions. Alternatively, Diffie-Hellman (DH) could analogously be used for the same purpose.

2.2.4 Sealing

Confidentially and integrity guarantees are provided during execution. Once destroyed, enclave data are lost. For this reason, SGX provides mechanisms for secure data persistence. Apart from storing application data, the sealing feature typically aids on certificates' storage, which obviates the need of new remote attestations every time an enclave application restarts. Like the keys used for attestation, sealing keys also depend on the platform hardware key derivation and are obtainable through the `EGETKEY` instruction.

Sealing key derivation can either consider `MRENCLAVE` or `MRSIGNER`. This choice determines who is able to decrypt the sealed data later on. If the enclave measurement is used, only the same enclave deployed on the same platform will be able to perform the unsealing. Any code modification would render inaccessible data sealed under this policy. This is useful when old data needs to be invalidated once new versions of the enclave are released, *e.g.*, due to a vulnerability mitigation. Instead, if the signer identity is used, every other enclave in the platform which was also signed by the same ISV is allowed to access the data. In this case, offline transfer of sealed data is possible as long as the same CPU performs the unsealing.

2.2.5 Platform services

Sealing preserves confidentiality of persistent data. It thus allows the storage of sensitive results and loading sealed secrets or certificates across enclaves re-initialisations. However, an attacker could still try to serve an enclave with previous versions of sealed data that are properly encrypted and authenticated. That can have severe consequences as the enclave could be misled to reflect some past state. To illustrate, imagine your bank account balance getting back to that of your pay day's eve. To prevent such replay attacks, an enclave can use monotonic counters provided by the platform. Each time an enclave writes a new state on disk, it increments a monotonic counter and stores the new value inside the sealed state. When the enclave restarts, it reads the monotonic counter and checks that it matches the value stored after unsealing the persistent data.

Due to the SGX restriction of limiting enclave executions to user-mode, trusted time and persistent monotonic counters are provided separately, as both require the usage of I/O either to access a clock or non-volatile memory. Intel decided to implement these services in the management engine (ME), a subsystem that allows remote system administration tasks like turning the machine on and off, regardless of having a running operating system. It consists of a proprietary firmware[1] running in a coprocessor in the chipset

[1] Minix OS, according to [53]

that is always active as long as the machine is connected to a power source. Because of that, these services may not be available in all platforms (particularly servers), as it depends on machine vendors.

To use such services, enclaves first need to open a session with a PSW enclave called platform services enclave (PSE) and have access to Internet [42]. Most likely, such connection provides a key that allows the PSE to communicate with the ME and then intermediate user enclave requests. Once the session is established, trusted timer and monotonic counters may be used. The trusted timer has a resolution of one second, which is substantial. It is mostly useful for determining the expiration of session keys or certificates rather than being used for profiling or performance measurements, since these often require finer grained timers.

Once an enclave requests the creation of a monotonic counter, it is given a universally unique identifier (UUID) associated to the newly created counter. This id must be remembered, so that the counter value may be incremented or queried. Similar to sealing, access control to counters may be associated to the enclave measurement (`MRENCLAVE`) or its signer (`MRSIGNER`). Such control is performed by the PSE.

Intel platform counters are known to be very slow: between $60\,\mathrm{ms}$ and $250\,\mathrm{ms}$ per increment, which are equivalent to alternative solutions such as TPMs [54]. Besides, since they are stored in flash memory they suffer from wear-out. Due to this, they can possibly stop working after a few hundreds of thousands write cycles [55]. Moreover, all counters are lost upon the re-installation of PSW [56] which can be done by privileged users, who are not trusted in the SGX threat model. Altogether, these reasons make Intel platform counters unsafe depending on security requirements. Distributed approaches are safer [56], although they may render poor performances for demanding applications. On the performance front, faster (and unsafe) layers backed by slower (and safer) ones may be used, although they are still vulnerable to rollbacks during the so-called *stability time* [57].

2.2.6 Vulnerabilities

A number of attacks to SGX enclaves were proposed. One class of attacks exploits software vulnerabilities like memory safety violations [58] (stack overflows or dangling pointers) to hijack the software control-flow [59]. Once that is accomplished, the attacker may find *gadgets* consisting in benign pieces of code (like the libc [60]) and reuse them to perform return-oriented programming (ROP) [61] in order to leak data or change the enclave behaviour. Such approach succeeds even on top of encrypted code provisioned to enclaves [62] or when countermeasure techniques like address space layout randomisation (ASLR) [63] are employed [64].

This kind of attacks lay down strong arguments for having small TCBs, as software bugs are proportional to code size. Nevertheless, several proposals provide complete runtimes in favour of usability, as they require little or no modification of legacy applications. Notable examples are Haven [65], SCONE [46], Graphene-SGX [66], Panoply [67] and SGX-LKL [68]. Indeed, software vulnerabilities fall out of the SGX threat model, which assumes to shield bug-free applications.

Another class of attacks consists of using alternative sources of information like power, sound, electromagnetic or, most importantly, time analysis to infer behaviour of targets and finally reveal some sensitive data that they handle. Processors nowadays are very complex machines and offer shared resources to several logical entities like VMs, OSes and processes. Usage of such components may leave traces that can be exploited by malicious agents [69]. One of these shared resources is the cache hierarchy, notably the most targeted component by SGX side-channel attacks [70]. Since the MEE sits at the edge of the on-chip memory hierarchy (see Section 2.2.1) and therefore inside the trust boundary, SGX cannot prevent software side-channel attacks that exploit cache leaks. Like for software correctness, Intel claims that "*preventing side channel attacks is a matter for the enclave developer*" [71] and, in fact, some solutions for preventing them were proposed [72, 73].

Recently, micro-architectural implementation bugs were exploited for breaching privilege barriers and getting access to the whole memory address space, including kernel pages. They are based on components that are supposed to improve performance like speculative execution [74, 75, 76] and out-of-order execution [77]. The biggest issue is that these attacks can leak sensitive data of perfectly secure bug-free applications. Foreshadow [78], for instance, was ultimately able to retrieve the long-term attestation signing keys from the SGX quoting enclave, making it possible to decipher all communication made by

an attested enclave through a man-in-the-middle attack. Countermeasures were released both in system software and microcode updates, although future microarchitecture versions are supposed to handle those issues in the silicon.

We also proposed an attack [6]. Instead of targeting running enclaves, we exploit the separation between building and signing stages of the supply chain. The attack is based on the assumption that a malicious agent is able to infect the machine where the enclave binary is signed. Once there, it suspends the signer process and diverts its input to a component that injects malicious code within the binary. The tampered version is then given back to the signer, which continues to execute as if nothing had happened, therefore giving authenticity to the infected binary. Every subsequent integrity and signature checks would then succeed. We also proposed mitigation measures by atomically shielding compilation and signing stages within enclaves.

Despite all the reported issues, throughout this work we explore Intel SGX in a range of distributed systems. We consider these flaws to be orthogonal to our research, and hence do not consider them in our evaluations. From an architectural vantage point, system designs that count on TEEs do not lose their relevance due to occasional breaches. In principle, they could use different (even future) hardware implementations that retain equivalent or comparable features to SGX.

2.3 SecureCloud project

A substantial part of this work was done in the context of the SecureCloud project [7] (2016-2018), whose goal was to enable the deployment of sensitive applications in the cloud while having strong security and privacy guarantees. That was built upon a layered architecture on top of SGX. Each layer provides a set of micro-services which can be combined into full-fledged cloud systems. The platform was validated with smart metering applications [79]. The consortium was composed by 14 partners in Europe, Israel and Brazil, including universities, research institutes, industry and government agencies.

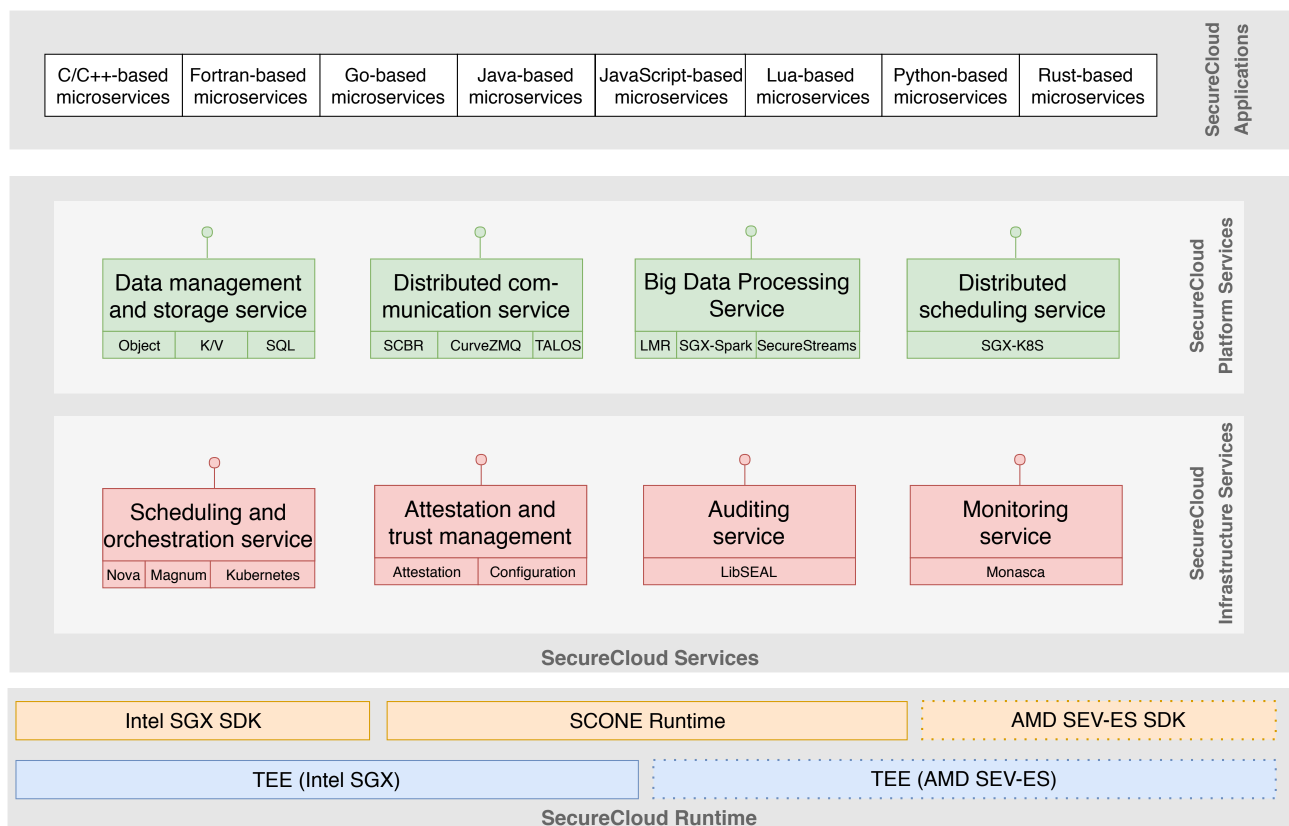

Figure 2.10: SecureCloud platform architecture.

Figure 2.10 shows the architecture of SecureCloud's platform, including SCBR (Section 3.1.1), LMR (*i.e.*, lightweight MapReduce—Section 3.2.2) and SecureStreams (Section 3.2.3). Besides, we also contributed with SGX-K8S [8], a SGX-aware solution for orchestrating containers in Kubernetes clusters. That was achieved by modifying the SGX driver to report the actual usage of EPC memory by each process to help on scheduling placement decisions. Moreover, the protected memory report was used for preventing containers from over-committing the EPC thus avoiding performance losses due to memory swaps (see Section 2.2.1) in the benefit of all other users who share the same platform.

Apart from our work, SecureCloud project produced a number of other relevant scientific contributions. The secure container environment (SCONE) [46], for instance, provides a complete runtime for deploying SGX containers. Glamdring [80] aids on partitioning SGX applications through code annotations. LibSEAL [81], in turn, allows auditing enclave operations and checking invariants to find integrity violations. More information about the project, along with deliverables, demonstrators and list of publications can be found in the official website [82].

2.4 Communication and data processing

This section provides background information concerning the Chapter 3. It describes system designs that fit in cloud deployment scenarios and can take advantage of TEEs. Specifically, Section 2.4.1 covers secure pub/sub systems while Section 2.4.2 and Section 2.4.3 discuss about distributed processing frameworks.

2.4.1 Secure publish/subscribe

Content-based routing (CBR) is a powerful communication model that supports scalable asynchronous communication among large sets of geographically distributed nodes [83]. Yet, preserving privacy represents a major limitation for the wide adoption of content-based routing (CBR), notably when the routers are located in public clouds. This represents a major deterrent for companies for which data is a key asset, as for instance in the case of financial markets.

In the pub/sub model, publisher nodes submit data to a routing service as publications formed of a header describing the data and a payload containing the effective data. The routing service matches the publications header with subscriptions previously registered by subscriber nodes and further routes the matching publications towards their destinations. Content-based pub/sub appears to be incompatible with privacy preservation, as messages must be filtered based on their content, *i.e.*, the routing engine should be able to see both the payload of publications and subscriptions. Additionally, by structuring the containment relations between subscriptions, one can build efficient data structures to store subscriptions and match publications. Containment allows for a significant reduction of the number of subscriptions stored as well as the number of matching evaluations executed per publication. As a consequence, containment is used in most CBR systems in use today [84, 85, 86], which makes them largely incompatible with classical cryptographic techniques for privacy preservation.

There is vast amount of literature about secure content-based pub/sub, including major surveys [87, 88], but no solution provides at the same time powerful filtering capabilities and high performance. While there exist some techniques for privacy-preserving computation, they are either prohibitively slow or too limited to be usable in real systems.

Specialised solutions, like asymmetric scalar-product preserving encryption (ASPE) [89], allow for a direct match of encrypted publications with encrypted subscriptions. Publication attributes and subscriptions constraints are represented as coordinates of multidimensional points. ASPE is based on an exact relation preserving isomorphism and supports subscription containment, although it is vulnerable to known-plaintext attacks. Given that ASPE's matching complexity is prohibitively high when using a large number of attributes, an alternative [90] was proposed to enhance it with a pre-filtering approach that expresses equality constraints using Bloom filters [91]. This allows for quickly identifying subscriptions that do not match the publication as their equality constraints cannot be satisfied.

Solutions based on traditional techniques encrypt sensitive information that traverse untrusted domains, hence basically preventing routers from filtering publications based on their content. They propose architectures designed to respond to specific security threats by combining access control and key management mechanisms. Most of these solutions integrate elaborated access control models into existing event-based middleware [92, 93, 94], organising routing brokers in sets that share keys to encrypt or decrypt data. Different encryption granularities are used, ranging from individual attributes to entire messages.

In Section 3.1.1, we follow a different strategy by taking advantage of TEEs. We implement a CBR engine in a secure enclave, so that its compute-intensive operations can work on decrypted data and leverage efficient matching algorithms. Our experimental evaluation shows that SGX adds only limited overhead to insecure plaintext matching outside secure enclaves while providing much better performance and more powerful filtering capabilities than alternative software-only solutions. Since its publication [10], further research explored our scheme [95] or proposed alternatives [96].

2.4.2 Batch processing

MapReduce is a programming model used extensively for parallel data processing in distributed environments. A wide range of algorithms were implemented using MapReduce, from simple tasks like sorting and searching up to complex clustering and machine learning operations. Many of these implementations are part of services externalized to cloud infrastructures, where concerns regarding security guarantees are common. In Section 3.2.2, we explore the use of Intel SGX for providing privacy guarantees for MapReduce operations, and based on our evaluation we conclude that it represents a viable alternative to cryptographic mechanisms. We present results based on the widely used k-means clustering algorithm, but our implementation can be generalized to other applications that can be expressed using the MapReduce model.

A number of security schemes for MapReduce were proposed. SecureMR [97] focuses on data integrity, verification of results and prevention of replay attacks, but it does not handle privacy of data and code. Airavat [98] and Gupt [99] combine differential privacy [100] with access control policies and therefore incur on the classical trade-off between utility and privacy, *i.e.*, too much noise produces bad results while no obfuscation renders no privacy. Through static code analysis, MrCrypt [101] selects distinct homomorphic encryption schemes for every data column. As expected, it is very slow: some benchmarks show performance penalties of one order of magnitude with respect to unencrypted execution. Sedic [102] and Tagged MapReduce [103, 104] split tasks to be deployed on private and public clouds based on how sensitive are the data they compute. Besides the requirement of partitioning applications and failing to handle the case in which all computation happens in shared infrastructures, they impose higher latencies due to data transfers between different clouds.

Closer to our approach, verifiable confidential cloud computing (VC3) [105] is a distributed, secure execution environment extending the Apache Hadoop MapReduce framework, originally implemented and evaluated in an SGX emulator. Each worker node hosts a trusted loader that runs a key exchange protocol in the enclave before decrypting and executing map/reduce functions. VC3 guarantees global integrity by generating verifiable work summaries within trusted workers. The user code is written in C++, which can make the implementations prone to potential faults like illegal memory accesses. In this regard, authors provide an optional compiler through which the programmers can enforce self-integrity invariants for memory regions.

An extension of VC3 appears in [106], which focuses on security issues generated by traffic analysis attacks on the exchanges between mapper and reducer nodes. We do not consider such attacks in our evaluation, but we believe that the analysis and the solutions proposed are also applicable to our system. M^2R [107] also presents a secure framework for MapReduce, which takes a more general approach with a design that can be implemented on any TEE. The authors refer to SGX-enabled processors as one potential target, but like VC3, the evaluation is conducted on simulated TEEs. As in [106], M^2R specifically focuses on attacks exploiting the leakage between mappers and reducers.

The general idea behind the above frameworks is close to our proposal in the sense that users can write their own *map* and *reduce* functions that execute in TEEs. However, we take a different approach with respect to the processing mechanism of interpreted code and to the data dissemination on top of secure

content-based routing (SCBR). Besides the fact that we evaluate on real SGX hardware, our approach provides additional flexibility and ease-of-use through the high-level Lua-based programming environment (see Section 3.2.2).

2.4.3 Stream processing

Stream processing represents another class of computations. It consists in the data treatment of continuous events, such as sensor measurements, user activity in social networks or financial transactions. Like batch processing, it is also subject to security and privacy concerns when deployed in hostile environments such as multi-tenant clouds.

Several industrial players introduced their own stream processing solutions, *e.g.*, Twitter's Heron [108] and Google's Cloud DataFlow [109]. These systems are mainly used to ingest massive amounts of data and efficiently perform real-time analytics. They are typically deployed on the provider's premises and are not offered as a service to end-users.

Some open-source middleware frameworks like Apache Spark [110], Apache Storm [111] and Infinispan [112] introduced application programming interfaces (APIs) to allow developers to quickly set up and deploy stream processing infrastructures. These systems rely on the Java virtual machine (JVM) [113], whose memory requirements impose considerable challenges to achieve good performance in the face of SGX constraints. Spark [114] is the most prominent solution. It leverages resilient distributed datasets (RDDs) to provide a uniform view on processing data. Despite its popularity, vanilla Spark only handles unencrypted data and hence does not offer security guarantees. Some proposals extend it to provide security for data at rest [115]. More recently, researchers ported it to run in SGX enclaves: Sgx-Spark [116] which is based on SGX-LKL [68] and SGX-PySpark [117] based on SCONE [46].

Some proposals rely on a hybrid model with trusted and untrusted infrastructures, where critical processing is done in private clouds. Under this model, Styx [118] uses partial homomorphic encryption and show overheads of $25\,\%$ in comparison to Apache Storm. However, they cannot guarantee data integrity. A few dedicated solutions exist today for distributed stream processing using reactive programming. For instance, Reactive Kafka [119] allows stream processing atop of Apache Kafka [120, 121]. Such solutions do not support secure execution in TEEs.

Section 3.2.3 introduces SecureStreams, a reactive middleware framework to deploy and process secure streams at scale by decrypting data inside enclaves and performing plaintext processing. Its design combines the high-level reactive data-flow programming paradigm with TEEs in order to ensure privacy and integrity of the processed data. The experimental results of SecureStreams are promising: while offering a fluent scripting language based on Lua, our middleware delivers high processing throughput, thus enabling developers to implement secure processing pipelines in just few lines of code. To the best of our knowledge, SecureStreams was the first lightweight and low-memory footprint stream processing framework that can fully execute within SGX enclaves. Later on, StreamBox-TZ [122] was proposed for stream processing at the edge with ARM TrustZone, which counts on weaker security guarantees (see Section 2.1.2).

2.5 Data sharing

Using public cloud services for storing and sharing confidential data requires mechanisms to cryptographically protect the data and limit the access to them. We use TEEs to tackle this subject in Chapter 4. In Section 4.1.1, we introduce IBBE-SGX, a new cryptographic access control extension that is efficient both in terms of computation and storage even when processing large and dynamic workloads of membership operations. IBBE-SGX addresses the impracticality of identity-based broadcast encryption (IBBE) by exploiting Intel SGX to derive cuts in the computational complexity of encryption operations. We also propose a group partitioning mechanism that attenuates the computational cost of decryption, so that it becomes bound to a constant partition size rather than the entire group's size. As a result, IBBE-SGX performs membership changes much faster than the traditional approach of hybrid encryption (HE) while producing less metadata.

In some cases, the identity of end users needs to remain confidential against the cloud provider and fellow users accessing the data. As such, the underlying cryptographic access control mechanism needs to ensure the anonymity of both data producers and consumers. We introduce A-Sky in Section 4.2.1, a cryptographic access control extension capable of providing confidentiality and anonymity guarantees. Thanks to TEEs, we are able to handle the impracticality of anonymous broadcast encryption (ANOBE) schemes with simple cryptographic constructs. We achieve faster execution times and shorter ciphertexts, being hence able to efficiently scale to large organisations.

We split the related work on *(i)* cryptographic schemes for access control, *(ii)* cryptographic protection for storage, and *(iii)* confidential messaging systems. We describe next what are the main features in each class of solutions and how they relate to our proposals.

Cryptography and access control

Hybrid encryption (HE) consists of using a public key to cipher a symmetric key. Once the resulting ciphertext is shared with an interlocutor who holds the corresponding private key, further communication is performed using symmetric encryption, which is faster and produces smaller ciphertexts. Such approach combined with a public key infrastructure (PKI) and identity-based encryption (IBE) was used in a role-based access control scheme. Its high overhead makes it unsuitable for reasonably dynamic cloud storage scenarios [123]. Attribute-based encryption (ABE) [124] is a cryptographic construction that allows a fine-grained access control by matching labelled attributes to users and content. Depending on the location of labels, one can distinguish between key-policy ABE [125] and ciphertext-policy ABE [126]. Even when employed for simple access control policies, ABE costs are substantially greater than IBE.

Hierarchical identity-based encryption (HIBE) [127] and functional encryption (FE) [128] are two cryptographic schemes offering functionalities for access control that rely on pairing-based cryptography, like IBE and ABE. HIBE is designed to handle hierarchical setups where each node may issue private keys to its descendants. FE is a powerful construct that can arbitrarily encapsulate programs as access control, but is unsuitable for practical use. Iron [129], which handles FE, is the closest to our proposals in the sense that it takes advantage of SGX to build a practical encryption scheme for an unpractical strategy thus far. Like IBBE-SGX, they use an enclave that holds a master secret as root for later key derivations. The enclave generates a key that is associated to a function, so that the computation can be performed without revealing the data on top of which it is applied. However, results of applying such function are presented in clear.

Proxy re-encryption [130] allows a data owner to delegate re-encryption to a proxy by sharing a transformational key with it, with the intent of sharing these data with another user. It can possibly be combined with IBE [131] or ABE [132, 133] and is suitable for cloud environments, as the re-encryption and data storage can happen in the same premises. A-Sky (Section 4.2.1) uses a similar proxying approach for writing operations, although we do not require users to share keys.

Multicast communication security [134, 135] defines efficient schemes that focus on revocation aspects. Logical Key Hierarchy [136] is a re-keying approach in which communications costs for revocation operations are logarithmic. Other schemes [137, 138] exploit secret sharing mechanisms that tolerate up to a maximum numbers of colluding revoked users.

Like HE, IBBE-SGX (Section 4.1.1) encrypts a shared key with a more complex cryptogrpahic scheme, *i.e.*, IBBE, so that simpler and faster approaches like AES can further leverage that key. A-Sky (Section 4.2.1), on the other hand, employs symmetric encryption to encapsulate the shared key. That is done, however, once per group member. Composing the groups in both systems is responsibility of an administrator who has no access to keys. We adopt a *lazy* revocation policy, *i.e.*, revoked users lose access to future group keys but keep being able to read old data. In our systems, collusion between any number of revoked users cannot grant access to keys encapsulated after their revocations.

Cryptographic cloud storages

A number of storage and sharing system designs have been proposed for mitigating the lack of trust in cloud providers. DepSKY [139] proposes a client-side object store that encrypts and redundantly stores ciphertexts on multiple untrusted storages. Encryption keys are split by using a secret sharing

Table 2.2: GPG latency in hidden recipient mode.

Group size	Avg. encrypt [s]	Avg. decrypt [s]	Envelope size [kB]
10	0.1	0.6	5.3
10^2	0.7	5.8	16.5
10^3	12	60	129

scheme [140] and dispersed over multiple storage premises that are assumed to not collude with each other. SCFS [141] extends DepSKY by using a trusted metadata coordination service that also encapsulates access control, which, in turn, is not cryptographically protected. This coordinator service is trusted and can be therefore compromised if an attacker breaches into it.

Other systems cryptographically enforce access control using *key envelopes*. CloudProof [142] is a cloud storage system with client-side encryption that solves access control by using broadcast encryption [143] to envelope two keys: one for encryption and another for signing. It offers confidentiality, integrity, freshness and write-serializability. CloudProof does not discuss how the authenticity of users' identity is established and checked. Hypothetically, PKI could be used, thus requiring a trusted entity in the system. To avoid that, IBBE-SGX relies on the identity-based version of broadcast encryption.

Sieve [144], instead, uses ABE for access control and key homomorphism for providing a zero knowledge guarantee against the storage provider. It allows users to store encrypted data in the cloud and to delegate access to consuming web services. REED [145] also uses ABE [126] to envelope symmetric keys that cipher deduplicated content. Performance overheads of rekeying operations drastically increase to several seconds when the number of users is as low as 500. This reinforces the inability of ABE's practicality for large scale access control.

Our protocols are agnostic to the storage system. Both IBBE-SGX and A-Sky cryptographically protect a key that can only be deciphered by rightful group members. To do that, these members need a ciphertext and additional metadata that could possibly be disseminated through any channel. We leverage the envelope idea in A-Sky, so that each encrypted file has corresponding metadata that are sufficient for retrieving its plaintext content. IBBE-SGX, on the other hand, has both a global piece of information (the public key) and individual pieces of metatada per group. In both systems, infrastructure providers learn nothing about keys.

Confidential messaging systems

Encrypted messaging systems share a common initial phase with our file sharing model by requiring the construction of a group key that shields group communication. Popular messaging systems (*e.g.*, WhatsApp, Threema, Signal) use a DH group key agreement and derivation [146]. Such protocols require all active participants to contribute to the creation of the group key, albeit with no anonymity guarantees. Pung [147] uses private information retrieval (PIR) in conjunction to a group DH key derivation, thus achieving anonymity. Such mechanisms are different from our target model, in which active users do not need to participate in the group key's creation.

Pretty good privacy (PGP) [148] is used for cryptographic protection of files or emails. It addresses anonymity (hidden recipient mode, in its nomenclature) by performing symmetric encryption of the content and then several public key encryptions of the shared key, one per user in the sharing group. An outside adversary only sees ciphertext and cannot infer who are the recipients. At decryption time, rightful recipients must try to decrypt each fragment until they succeed ($\frac{n}{2}$ trials on average, where n is the group size). We took a similar approach with A-Sky (Section 4.2.1), but using symmetric cryptography inside enclaves. Table 2.2 shows the results of a simple benchmark of GNU privacy guard (GPG) v. 1.4.2 in hidden recipient mode. Encryption and decryption have enormous latencies of $12\,\mathrm{s}$ and $60\,\mathrm{s}$ for groups of 1000 members, respectively. Moreover, the inner implementation of PGP's hidden recipient mode is reputed as insecure against chosen ciphertext attacks [149].

The problem of devising a cryptographic scheme that can guarantee both confidentiality and anonymity is referred to as *anonymous* (or *private*) *broadcast encryption* (ANOBE). A few of such schemes were

proposed, however without assessing their practicality within real systems. Barth *et al.* [149] (*BBW*, per authors' initials) achieves inner and outer anonymity, in addition to guaranteeing indistinguishability under non-adaptive and adaptive chosen ciphertext attack (IND-CCA). It extends the public key enveloping model of PGP by incorporating signatures [150] such that an attacker who is a group member cannot reuse envelopes to broadcast arbitrary messages to the group. *BBW* can handle key enveloping throughputs of only few hundreds of users per second.

Additionally, they propose the construction of publicly-known labels to decrease the number of decryption trials. The ciphertext fragments in the envelope are sorted by labels and can therefore be located in logarithmic time at decryption. This way, only one asymmetric deciphering operation is performed. The scheme was further extended by Libert *et al.* [151] by suggesting the use of *tag-based encryption* [152] to hint users about their ciphertext fragment.

A-Sky provides indexing mechanisms for achieving better performance at decryption time, so that the specific ciphertext of a given user can be located inside the envelope in logarithmic time. Like in PGP, A-Sky also encrypts the same data several times, once per group member. We use instead more efficient symmetric cryptographic algorithms. Unlike the messaging systems mentioned above, key generation in both IBBE-SGX and A-Sky is made inside enclaves and is simply based on random number generators. Therefore, such procedure does not require that individual users be online to contribute in the constitution of shared keys.

2.6 Privacy assurance

Not every system can be designed from scratch to properly ensure security and privacy. Sometimes users must use established services that are unlikely to change, specially if monitoring user activity is at the heart of the service provider's economic model. That is the case of Web search engines whose business depend on targeted advertisements. Chapter 5 discusses privacy-preserving systems that use TEEs as building blocks.

By regularly querying Web search engines, users disclose large amounts of their personal data as part of the queries, possibly unconsciously. Among these data, they may reveal sensitive information like health issues, sexual, political or religious preferences. Nowadays, there is no satisfactory approach to enable users to access search engines in a privacy-preserving way. Existing solutions are either subject to attacks, too costly due to heavy use of cryptographic mechanisms, dependent on weak adversarial models or unable to provide accurate results.

Privacy-preserving proposals can be classified in three main categories. One of them is composed of alternative search engines which implement specific privacy-preserving protocols generally based on PIR, thus enforcing privacy by design. In such systems, users access data stored on the remote server without revealing what information they access. In general, PIR protocols consist of three algorithms: building protected queries with encrypted keywords, performing the information retrieval in such a way that the search engine has no access to the query and results, and finally reconstructing the result list on the client side. They generally rely on heavy cryptographic protocols that are still unpractical [153], especially when data stores contain millions of documents, which is the case of search engines.

The other two categories of privacy-preserving systems enable clients to use existing search engines while offering them a set of privacy guarantees. In the text that follows, we describe related work in private Web search and discuss the limitations of existing solutions.

Enforcing unlinkability

This category of solutions includes a set of protocols enabling users to anonymously send queries to a search engine, thus enforcing unlinkability between queries and the identity of their issuing users (*e.g.*, IP address). The most popular protocol among these solutions is the onion router (Tor) [154], an implementation of the onion routing protocol [155]. Tor (Figure 2.11) sends each query through randomly-selected nodes using a cryptographic protocol where queries are encrypted using a public key of each node in the chain. The ciphertext can hence be associated to an *onion*, *i.e.*, with multiple layers. Each relay node deciphers the outermost layer of the onion and further forwards the remaining blob until it reaches the

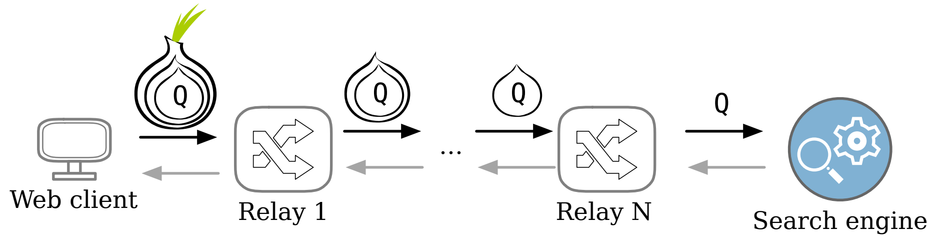

Figure 2.11: Unlinkability: Tor.

exit node. The exit node, in turn, can finally send the query to the search engine on behalf of the original user. One of the limitations of this protocol is that participating relays are assumed to faithfully execute the forwarding of onions, which might not be true as some may behave selfishly, *e.g.*, by dropping them, or even maliciously, *e.g.*, by injecting fake traffic to slow down the system.

To mitigate negative effects caused by selfish or malicious users, RAC [156] was proposed. In this protocol, nodes are organised on several virtual rings in a way that for each ring a node has a predecessor and a successor node. A node might be part of several rings and thus have multiple predecessors and successors. To ensure that no message is dropped by a freerider, nodes have to broadcast all messages they relay. If a node does not receive a message from a given predecessor, it is considered as a freerider. RAC suffers from severe performance limitations, presenting throughputs that are orders of magnitude lower than Tor. The dissent protocol [157, 158], in turn, enforces accountability in presence of malicious and selfish participants. However, its performance is even worse than the one of RAC as it is a combination of two heavy cryptographic protocols [159, 160].

In addition to the performance issue, protocols enforcing unlinkability have been shown not to resist to re-identification attacks [161, 162]. Assuming a set of user profiles built from user past queries, user re-identification attacks try to link anonymous queries to a profile corresponding to their originating user. The issue comes from the fact that search queries themselves disclose enough information for breaking the unlinkability property. The reason is that users tend to look for similar things even when they have a different Internet protocol (IP) address. Besides, browser metadata included in the hypertext transfer protocol (HTTP) headers can also help to identify users.

Unlikability is enforced in both systems we propose in Chapter 5. While X-Search (Section 5.1.1) hides the identities of users from search engines by proxying their requests, Cyclosa (Section 5.2.1) creates a peer-to-peer (P2P) network of users who also act as relays. Despite the flaws we listed above, service providers cannot tell with certainty where requests come from if the endpoint is not really the issuer, unless they have additional information. This is why we further try not to provide them with this supplementary data, as we discuss in the next section.

Enforcing indistinguishability

Another class of solutions aim at making real user interests indistinguishable from fake ones. QueryScrambler [163] protects users by replacing their requests by semantically related queries. For each request, it generates a set of related queries by generalizing the concepts used in the initial query. By merging and filtering all the results obtained with these related queries, it retrieves the most plausible results for the initial query. Unfortunately, results' accuracy might be impaired.

Figure 2.12: Indistinguishability: *(a)* TrackMeNot and *(b)* GooPIR.

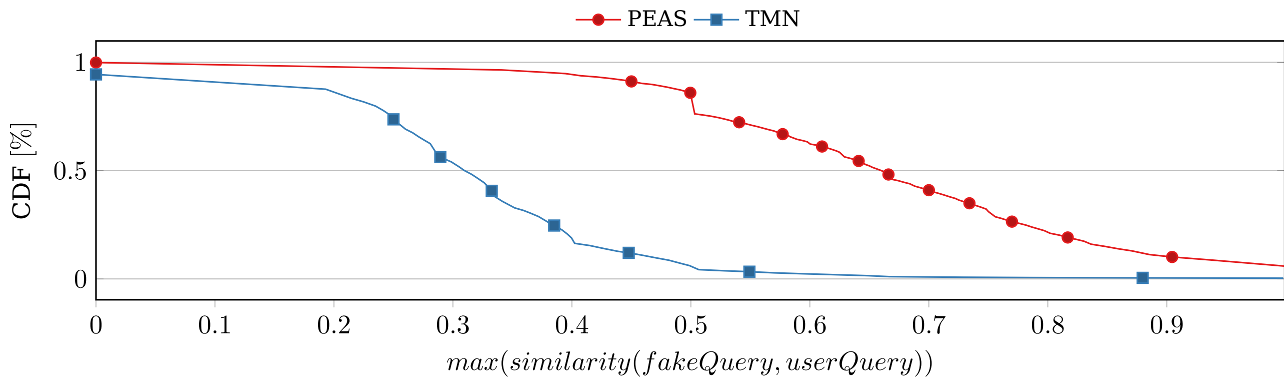

Figure 2.13: Similarity between fake queries generated by TrackMeNot and PEAS with the AOL dataset.

TrackMeNot [164] (Figure 2.12a) is a browser extension which periodically sends fake queries to the search engine on behalf of users, independently of their real queries. The intent is that the user profile stored at the search engine will eventually get obfuscated by mixing the user's real interests with fake ones. Instead, GooPIR [165] (Figure 2.12b) obfuscates each user query by aggregating $k-1$ fake queries with the real one using the logical `OR` operator. As such, the search engine cannot distinguish the real query from fake ones. However, these solutions suffer from two limitations:

(i) the user's identity is still known to the search engine; and

(ii) they are subject to attacks as the fake queries they generate (based on RSS feeds in TrackMeNot and dictionaries in GooPIR) are easily distinguishable from real ones.

To overcome these limitations a solution named PEAS [166] (Figure 2.14a), combining both unlinkability and indistinguishability was proposed, based on two non-colluding servers. The first, called *proxy*, has access to the requester's identity but not to the query's content since it is encrypted with the public key of the second server. This server, in turn, is called *issuer* and has access to the query but does not know the originating user. In addition to forwarding the query on behalf of the user, the issuer generates $k-1$ fake queries and aggregates them with the original query to enforce indistinguishability. Differently from GooPIR and TrackMeNot, PEAS's fake queries are generated using a co-occurrence matrix of terms built by the issuer from other users past queries. Hence, PEAS better resists re-identification attacks as its fake queries are syntactically closer to real ones.

To highlight how challenging it is to generate efficient fake queries, we show in Figure 2.13 the cumulative distribution function (CDF) of the maximum similarity between fake queries generated by PEAS (*i.e.*, based on the co-occurrence of terms in past queries) and TrackMeNot (*i.e.*, based on RSS feeds) and past queries on the America online (AOL) dataset [167] (the similarity metric is further detailed in Section 5.1.1). Even though PEAS produces better fake queries (median similarity of 0.65 against 0.3 of TrackMeNot), most of the fake queries are significantly different from real ones, *i.e.*, they have never been requested to AOL.

Section 5.1.1 introduces X-Search (Figure 2.14b), a novel private Web search mechanism that builds upon SGX for proxying user requests. X-Search enforces both unlinkability and indistinguishability. It runs query obfuscation based on past queries on untrusted proxy nodes within enclaves. Apart from improvements on performance, it operates under a stronger adversarial model than its alternatives and better resists to re-identification attacks.

One of the above solutions' limitations is that all user queries get obfuscated with the same intensity, *i.e.*, by generating $k-1$ fake queries in GooPIR, PEAS and X-Search regardless of their sensitivity. As a consequence, a small value of k may lead to under protecting sensitive queries, which increases the risk that they get linked back to the original user. Conversely, a large value of k may unnecessarily generate large amounts of traffic. Cyclosa (Figure 2.14c), presented in Section 5.2.1, handles this issue while being scalable and returning accurate results. It is fully decentralised, spreading the load for distributing fake queries among other SGX peers. The number of fake queries used for obfuscation is dynamically adapted

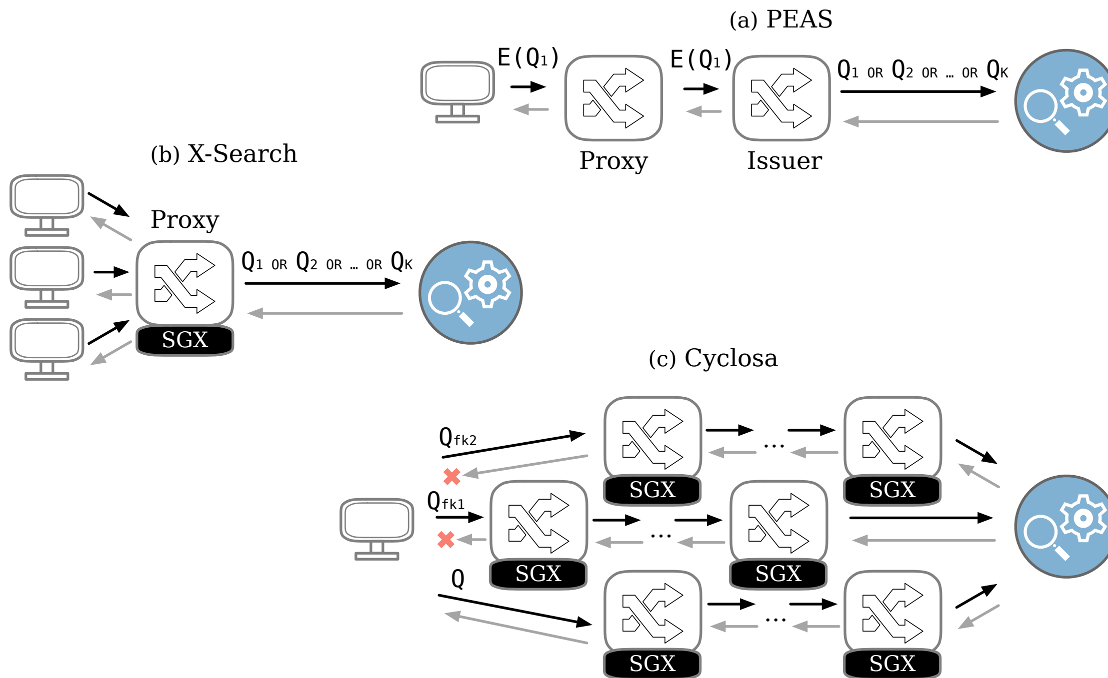

Figure 2.14: Indistinguishability and unlinkability: *(a)* PEAS, *(b)* X-Search and *(c)* Cyclosa.

to the user query's sensitivity. As it handles the real and fake queries separately, it achieves perfect results' accuracy.

Wrapping-up private web search

With regards to unlinkability, existing protocols are either efficient but assume honest but curious servers or robust to malicious adversaries but have unpractical performance (*e.g.*, Dissent, RAC). In terms of indistinguishability, the challenge is to generate realistic fake queries that are as close as possible to real ones.

Enforcing indistinguishability by aggregating the user query with fake queries (*e.g.*, using the `OR` operator) generates noise in the responses sent by the search engine as the results corresponding to fake queries get merged with those related to the real one. This noise is generally filtered out at the client side (in PEAS and GooPIR) or by the proxy (for X-Search) by removing the responses that do not contain words composing the original query. Despite this, relevant responses of the original query may be lost while noise may be returned to the user. Furthermore, the logical `OR` operator for multiword-based queries is not natively supported by all search engines.

Practical private Web search mechanisms must scale. This is not the case of centralised schemes such as PEAS or X-Search. In addition to the ability of private Web search to sustain the load coming from Internet users, a more concrete problem comes from the rate limitations imposed by search engines to counter bots and DDoS attacks. Google's bot protection, for instance, triggers after about 1000 queries before either asking to fill a captcha or refusing the request. Another problem of approaches like PEAS and X-Search is the monetary cost of deploying proxies. In contrast, Cyclosa leverages client machines that require no deployment. Table 2.3 summarizes the comparison between private web search solutions.

Web and SGX

TrustJs is a framework for trustworthy execution of security-sensitive JavaScript code inside commodity browsers [168]. It leverages enclaves to protect the client-side execution of JavaScript, enabling a flexible partitioning of web application code. Being attested by the server, the trusted interpreter can be used to offload its computation, which results in lower latencies in the user experience and lower performance demand for the application servers.

Table 2.3: Comparison of private Web search mechanisms.

	Tor	TMN	GooPIR	PEAS	X-Search	Cyclosa
Unlinkability	✓	✗	✗	✓	✓	✓
Indistinguishability	✗	✓	✓	✓	✓	✓
Accuracy	✓	✓	✗	✗	✗	✓
Scalability	✓	✓	✓	✗	✗	✓

Kim et al. [169] explored the possibility of using enclaves to provide security and privacy in network applications. They initially demonstrate how to use enclaves to prevent software-defined inter-domain routers to disclose their routing policies. They also show how the Tor anonymity network [154] can be strengthened by using enclaves to run its directory authorities to attest each other. Attackers can still launch denial-of-service attacks but they cannot alter the directory behaviour. Also, by putting onion routers within enclaves, they can attest their integrity and their admission can be done automatically so directory authorities can be eliminated, and the routers can simply keep track of their membership in a distributed hash table. Finally, they present how enclaves can be used to securely introduce in-network functionality into TLS sessions.

Cyclosa resembles TrustJs in its client-side nature. Rather than processing scripts locally though, we forward user queries to the P2P network of relays. With regard to Tor, Cyclosa is similar in the sense of establishing the network of collaborating users. The difference, besides the usage of symmetric encryption instead of layered ciphertexts, is that the enclave execution ensures fair behaviour. As the P2P network establishment is outside the scope of Cyclosa, we do not discuss about peer directories. Nevertheless, the aforementioned strategies in this regard could be employed.

2.7 Summary

In this chapter, we presented several commercially available TEEs and gave a general overview of how they operate. Special emphasis was given to Intel SGX. Due to its ensemble of security guarantees currently unmatched by alternatives, SGX is the TEE technology we further explore throughout the rest of this dissertation.

To contextualise part of this work's achievements, we briefly described the SecureCould project in Section 2.3 before covering the related work. In Section 2.4 we gave a general overview of potential cloud services materialised by communication and data processing frameworks, as per our contributions in this domain presented in Chapter 3. First, in Section 2.4.1, we covered security and privacy in a pub/sub middleware and next, batch (Section 2.4.2) and stream processing (Section 2.4.3) frameworks.

Section 2.5 outlined research work in cryptographic schemes that can be used for group data sharing in untrusted channels. We discussed access control, cloud storages and messaging systems related to our work detailed in Chapter 4, where we design innovative cryptographic protocols that count on TEEs for achieving better performance and security guarantees.

Finally, Section 2.6 explained the state of the art in private Web search. It described the concepts of unlikability and indistinguishability while presenting advantages and drawbacks of systems that enforce them. Doing so, it gave grounds for the contributions we report in Chapter 5.

Chapter 3

Communication and processing

To reach our goals, we start with empirically exploring the usage of trusted execution environments (TEEs) by designing, implementing and evaluating systems that naturally fit into cloud deployment scenarios, where many potential threats are possible (see Chapter 2). Particularly, we describe the implementation of a secure content-based routing engine in Section 3.1.1 and two distributed processing frameworks: one for batch executions, in Section 3.2.2, and another for handling event streams in Section 3.2.3. The goal is to quantify the software guard extensions (SGX) performance implications as discussed in Section 2.2.2 in the context of practical systems, besides learning about the main design concerns involved in building secure applications with the aid of TEEs. Results show that there are performance penalties whenever cache and enclave page cache (EPC) limits are surpassed. However, horizontal scalability helps to overcome the impact of memory constraints. In addition, the ability to replace complex cryptographic primitives with simple ones allows considerable gains in processing time.

3.1 Communication

The choice of content-based routing (CBR), a flexible paradigm for scalable communication among distributed processes, conforms well to our target scenario because it handles sensitive data and is well suited for off-site deployment. CBR decouples data producers from consumers by dynamically routing messages depending on their content. For this purpose, routers must filter messages (publications) by matching their content against a collection of stored predicates (subscriptions). Such scheme thus requires the router to see the content of both publications and subscriptions, which represents a considerable privacy threat. In the canonical example of stock market, for instance, quotes published by exchange platforms have commercial value, while subscriptions may reveal the client's interests and portfolio. Both the publisher's assets and user's privacy are at stake, hence why their sensitive information must be protected from leakage.

Since interacting processes are geographically scattered and routers are supposed to be deployed somewhere in between data producers and consumers, it makes sense to deploy them in third-party infrastructure providers. Moreover, they could serve several organisations (data providers), each of which having their own clients. All combined, we have the main elements for reinforcing the security of a system: third-party servers, multi-tenancy and sensitive data processing.

In this section, we describe an original CBR architecture that exploits SGX for executing a routing engine in a secure enclave. We also propose a protocol for securely exchanging keys among data producers, consumers, and the routing engine in the enclave. Publications and subscriptions are encrypted and signed, thus protecting the system from unauthorised parties willing to observe or tamper with the information. Our system, called secure content-based routing (SCBR) [10], thus combines a key exchange protocol and a state-of-the-art routing engine to provide both security and performance while executing under the protection of the secure enclave. We then evaluate our implementation with a few workloads to observe the sources of performance overheads and the various trade-offs of SGX. We provide comparative results against plain-text (insecure) matching as well as with an encrypted filtering alternative. To the best of our knowledge, SCBR was the first system to experimentally evaluate and demonstrate the benefits of executing CBR in a TEE.

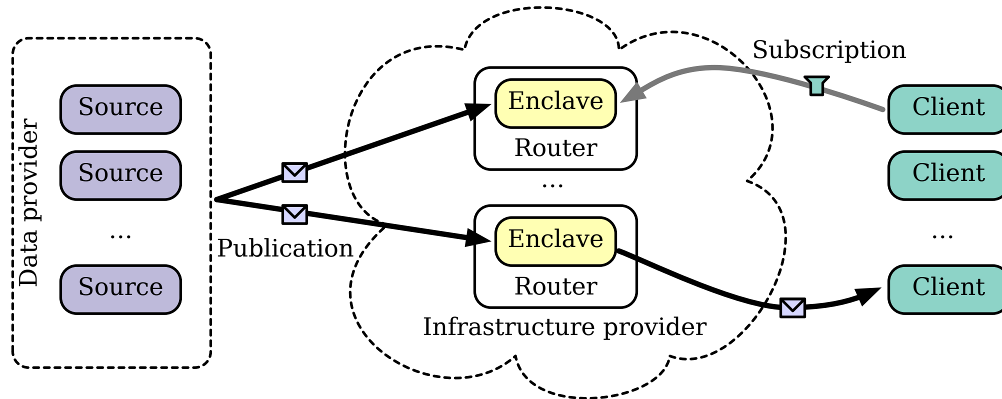

Figure 3.1: SCBR overview.

3.1.1 Secure content-based routing

Even if one can rely on trusted environments, designing a secure, privacy-preserving CBR system is not trivial. Consumers who want to protect their confidential data have to trust the code of the CBR engine. If it handles multiple publishers, from different administrative domains, each of which with their own clients, consumers might be discouraged of using the system fearing that their private subscription data could be accessible by other organisations. Besides, that would require the maintenance of multiple data structures within routers, one per group of data provider and their clients. Although technically feasible, we consider a simpler model based on the following assumptions:

- The publish/subscribe (pub/sub) system operates as a service under the control of a single “data provider” that publishes data.
- Consumers are the clients of the service and typically pay recurring fees to have access to the data.

Producers must be able to control which subscribers can join and read data from the service, and to exclude clients who stop paying their fees or give any other cause for contract dissolution. The publishers operate within the administrative domain of the service provider from which data originates, and they are trusted by the clients for the purpose of the considered service. This model closely maps, for instance, to the aforementioned scenario of a stock exchange.

Given the trust relationships between different components of SCBR, it is clear that publishers and clients must share cryptographic keys that are not known by the infrastructure provider. Furthermore, there should be some mechanisms for the publishers to include and revoke clients. Excluded clients should be prevented from receiving new data, independently of whether they previously were legitimate customers.

System Model

Following the considerations discussed above, we distinguish three main roles in our architecture, as illustrated in Figure 3.1:

- The *data provider* produces information streams for the clients, typically “as a service” and for a fee. The data may be produced by multiple sources (publishers) operating within the same administrative domain.
- The *infrastructure provider* hosts the CBR engines in the cloud. It provides secure hardware and performs the actual data routing and transmission through its network. As it operates under a different administrative domain and may share its resources among several customers in a multi-tenant configuration, the infrastructure provider is not trusted, although it accordingly performs its services.
- The *clients* of the service are the end users who are interested in the actual data and subscribe to information flows via the CBR engines. They trust the data providers but not the infrastructure.

Messages are composed of a payload, which is of interest to the end users but opaque to the router, and a header that contains several *attributes* and associated values. Since SCBR is not concerned about payload contents, its encryption is out of scope of this chapter. We discuss alternatives for encrypted group communication in Chapter 4. The SCBR router filters publication messages based on the attribute values in their header.

Subscriptions are composed of *predicates* specifying constraints over the attributes. Predicate expressions can include equality constraints or any kind of ranges over the attribute values, *i.e.*, they can use the operators $=$, $>$ or $<$. For instance, a subscriber interested in specific quotes for a company when they fall below a certain price can register a subscription such as "$symbol = \texttt{"ABC"} \wedge price < 40$". A publication message *matches* a subscription if its header satisfies the constraints expressed in the subscription predicate.

Subscriptions are typically stored by the CBR engine in a dedicated data structure that operates as an *inverted* database. Rather than actual data, the queries are stored instead, and data occasionally comes to be matched against them. By exploiting relationships between the different predicates [84], one can both reduce the memory footprint of the subscription index and improve the matching speed. In particular, the property of *containment* (or coverage) can be leveraged to avoid unnecessary tests. Essentially, we say that a subscription s contains or covers another subscription s' if any event that matches s' also matches s. That is, s is more general than s'. For instance, predicate "$x > 0$" covers both predicates "$x = 1$" and "$x > 0 \wedge y = 1$". Note that the containment relationships create a partial order on subscriptions that can be represented as a directed acyclic graph (DAG). In SCBR, we use a matching algorithm that exploits containment to minimise the footprint of stored subscriptions. This is particularly useful in enclaves, where only a limited amount of memory is available.

SCBR makes use of both symmetric and asymmetric (public key) cryptography. The former is more efficient and is used for communication between the publishers and the routers, while the latter is used between clients and the service provider when registering subscriptions, as will be detailed next.

The subscription process

We designed the system so that producers are the owners of the generated data. They have therefore the ability to decide whether they accept a subscription from a client, as well as to subsequently remove it. To control access to the service, we rely upon an additional admission phase when registering a new subscription. The client cannot freely submit its subscriptions to the CBR engines in the cloud, but has to go through a data producer. The registration process works as follows (see Figure 3.2).

Consider a client c that wants to register a subscription s by the routing engine r and subsequently receive message with header h sent by the data producer p. The publisher has a public/private key pair (PK/PK^{-1}), as well as symmetric key (SK) that is shared with the router running in the enclave, but unknown to clients and to the infrastructure provider. This is made possible thanks to SGX. Particularly, the symmetric key exchange happens during the attestation phase, as explained in Section 2.2.3.

1. The client first encrypts its subscription s using the data provider's public key, hence preventing unauthorised parties to see it, and sends the resulting encrypted subscription $\{s\}_{PK}$ to p.
2. Then, after decrypting and verifying that the subscription is valid, as well as verifying the client's status, the publisher re-encrypts s using SK and signs it. It then sends the encrypted subscription $\{s\}_{SK}$ to the routing engine r.
3. Finally, r validates and decrypts the subscription inside the enclave and inserts it in its index.

Subscriptions also embed location information about the clients that it visible to the code running outside the enclave. This allows the router to correctly address the forwarding of publications payload to matching consumers.

The publication process

Once subscriptions have been registered by the routers, data can flow along the reverse path. The publication process works as follows.

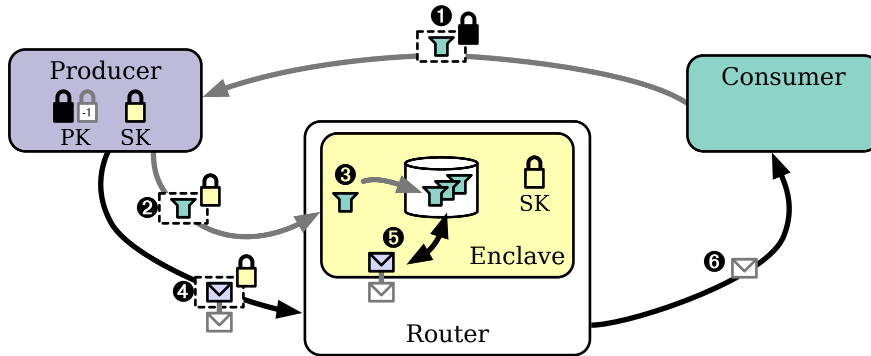

Figure 3.2: SCBR messages cycle.

4. The publisher encrypts the header h of the message using SK, which is only known to the code running inside the enclave. The encrypted message $\{h\}_{SK}$ is then sent to the routing engine r.

5. Upon receiving the message, r decrypts the header in the enclave, leaving the opaque payload outside, and matches it against its subscription index. The result of this operation is a list of clients that have registered a matching subscription.

6. Finally, r forwards the encrypted message payload to all clients that have been identified as part of the matching operation.

The payload of messages is encrypted separately. There are different approaches for doing so. A simple solution would be to use a symmetric group key shared between the publisher and all its consumers. Depending on the number and turnover of clients, this can incur in excessive communication overhead since each revocation would require a new key to be propagated across the set of users. This would allow publishers to prevent clients that have cancelled their membership from accessing newly published messages. This process is orthogonal to the privacy-preserving CBR, *i.e.*, the encryption performed for protecting the publications' header and subscriptions. We further discuss cryptographic protocols for group communication in Chapter 4.

Having multiple routers in the path would increase the complexity of the key management between publishers and matchers. We believe that an overlay broker network is not the best architecture for a scalable privacy-preserving pub/sub engine, and we would rather advocate for a similar structure to StreamHub [170], where system components are specialized in order to enhance performance. In such an architecture, the current publisher-router key management scheme could be simply replicated.

Evaluation

We evaluated the system as described in Figure 3.2, with both the producer and consumer running in one machine and the filtering engine in another. Measurements were collected at the machine running the filter, which was equipped with an Intel Skylake central processing unit (CPU) model i7-6700 running at $3.4\,\mathrm{GHz}$ with an $8\,\mathrm{MiB}$ cache and $8\,\mathrm{GiB}$ of main memory. We allocated $128\,\mathrm{MiB}$ of main memory to EPC (maximum allowed). In SCBR, encryption outside the enclave was implemented using the Crypto++ library [171] using respectively advanced encryption standard (AES) in counter mode (CTR) and Rivest-Shamir-Adleman (RSA) respectively for symmetric and asymmetric encryption. We use the ZeroMQ library [172] for communication, and we encode messages in Base64 text format. Information about page faults is obtained via the Linux system's `getrusage` function (attribute `minflt`). Similarly, we rely on a Linux system call to configure and read the processor's performance counters for cache misses.

To evaluate SCBR and facilitate comparison, we reused the workloads from previous work [90] by picking 3 out of the 9 datasets used then. They were chosen based on the diversity of performance output when applying our containment-aware matching algorithm to each dataset (Figure 3.4). They were built based on real data corresponding to randomly selected stock quotes from the Yahoo! finance website [173]. Approximately 250,000 entries were collected in a period of 5 years, with publications composed by 8 to 11 attributes. The entries collected were used to produce synthetic subscription datasets containing

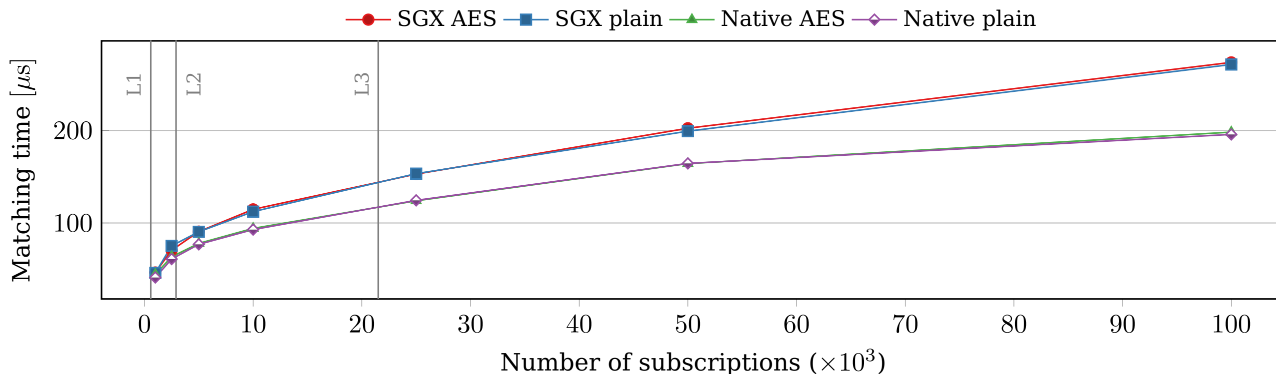

Figure 3.3: Comparison between native and SGX executions, with and without encryption.

an assortment of equality and range predicates on the quotes' attributes according to a uniform random distribution. In order to assess the algorithms' performance with a greater number of variables and different levels of containment, one workload was synthesised with four times the number of attributes of the original publications, by merging data from multiple quotes. Table 3.1 summarises the characteristics of the datasets used.

Our first experiment aimed at evaluating the performance overhead caused by executing our filter inside an enclave. We filled the subscription database with datasets with up to 100,000 subscriptions, reaching a total memory size of approximately 43 MiB. Then we sent a batch of 1,000 publications to be matched against the subscriptions and measured the time it took to accomplish each filtering operation. We ran an identical set-up with and without encryption, inside and outside an enclave, using the same filtering code. When using encryption, publications and subscriptions were encrypted in the producer and decrypted in the filter using AES-CTR. The average results for the first workload (*e100a1*) are shown in Figure 3.3. By considering the proximity of the lines with and without encryption, we can see that encryption overhead is small and nearly constant. Indeed, this overhead remains below 5 μs for both SGX and native executions, which is negligible when compared to the matching time given a reasonably large database size. The overhead resulting from the enclave is more significant, reaching nearly 40% for the largest set of subscriptions considered in this experiment, which is explained by enclave mode transitions and the occurrence of cache misses. Cumulative cache sizes are shown by the vertical lines.

We then focused on the influence of the workloads. In order to understand the effect of different datasets on SCBR performance, we first executed each of them without encryption outside secure enclaves. Results are shown in Figure 3.4. The first (*e100a1*) workload show the best performance, since all subscriptions contain equality predicates and the subscription set forms deeper containment trees. In contrast, datasets with more attributes (*e80a4*) perform worse because they yield indexes with more roots and shallow trees, therefore inducing more comparisons to traverse the whole subscription graph.

Figure 3.5 displays separate measurements for each workload running SCBR inside and outside an enclave, both using AES encryption. We also measured, for each workload, the performance of our implementation of asymmetric scalar-product preserving encryption (ASPE) [89, 90] as a baseline for a software-only alternative that does not use enclaves. We measured only the matching step, and not the encryption or decryption of ASPE messages. The presented ASPE performance cost was therefore inherent to its

Table 3.1: SCBR: Workloads description.

Workload name	Proportion of equality predicates	Number of attributes
`e100a1`	100% : 1 eq. pred.	8–11 (original)
`e80a1`	20% : 0 eq. pred.	
`e80a4`	80% : 1 eq. pred.	4× more

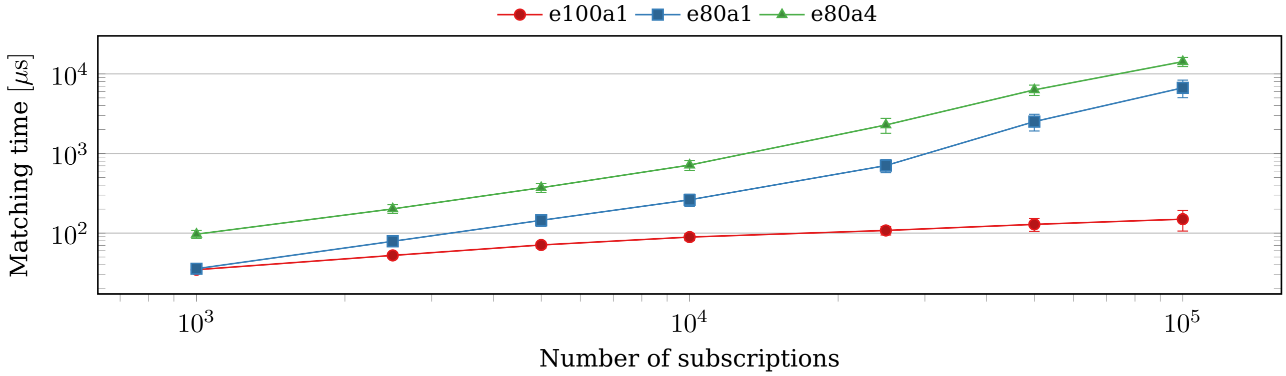

Figure 3.4: Performance of the containment-based algorithm applied to the different workloads in plaintext, outside enclaves.

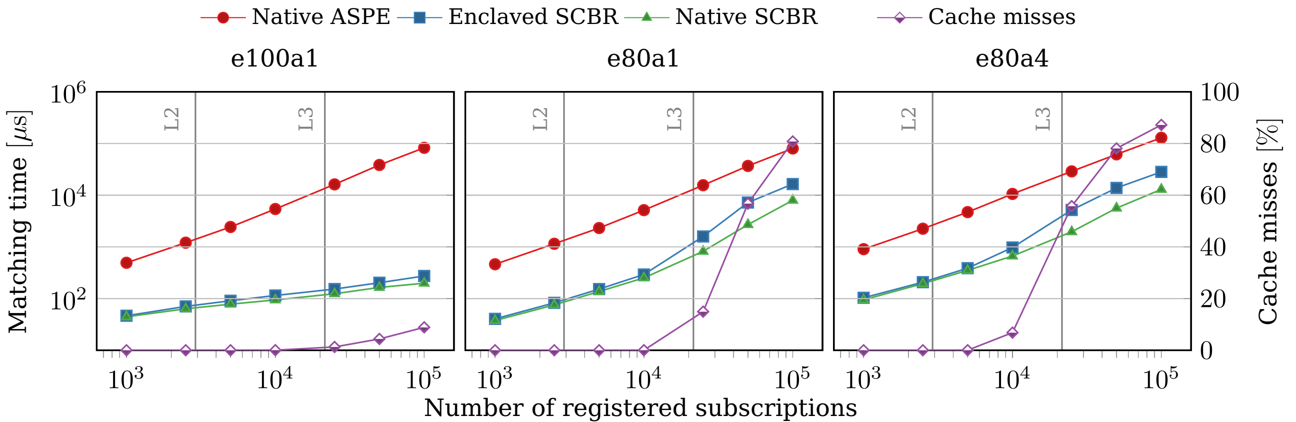

Figure 3.5: SCBR response to different workloads.

matching algorithm, which grows faster than any other strategy when increasing the size of the subscription database. The difference is more substantial for the first workload, although it remains close to at least one order of magnitude in all setups. These observations indicate that the performance penalties of SGX are largely tolerable when considering software-only alternatives for secure filtering, at least when the amount of memory used by the routing engine remains below the EPC assigned size.

Another interesting aspect is the gap between the curves corresponding to native and SGX executions. After approximately 10,000 subscriptions, the versions inside and outside enclaves begin to drift apart due to the number of memory accesses necessary to accomplish every comparison. At some point, the filtering data does not fit completely in the processor's cache memory and cache misses start to occur more frequently. When this happens, data must be fetched from system memory and, in the case of enclave executions, it must be decrypted and checked for integrity and freshness. Moreover, the evicted enclave's cache node must be encrypted before being sent to system memory. That behaviour is consistent with cache miss rates (measured outside the enclaves), which are also reported in Figure 3.5. We only measured cache misses outside the enclaves, because our Linux version failed to properly monitor the cache performance counters inside enclaves. Since the code running inside and outside the enclaves is the same, it is reasonable to assume that cache miss rates would be similar.

We finally observe the performance penalties when exceeding the maximum protected memory size and memory swapping begins to happen. Since EPC memory is limited, whenever it is full and more space is required, pages must be evicted from the protected area to the main (untrusted) memory. Accordingly, a page swap occurs every time a previously evicted page is accessed. Besides the fact that system memory is slower than the processor's cache, which already imposes performance costs, memory page swaps are serviced by the operating system (OS) and hence incur an even higher overhead.

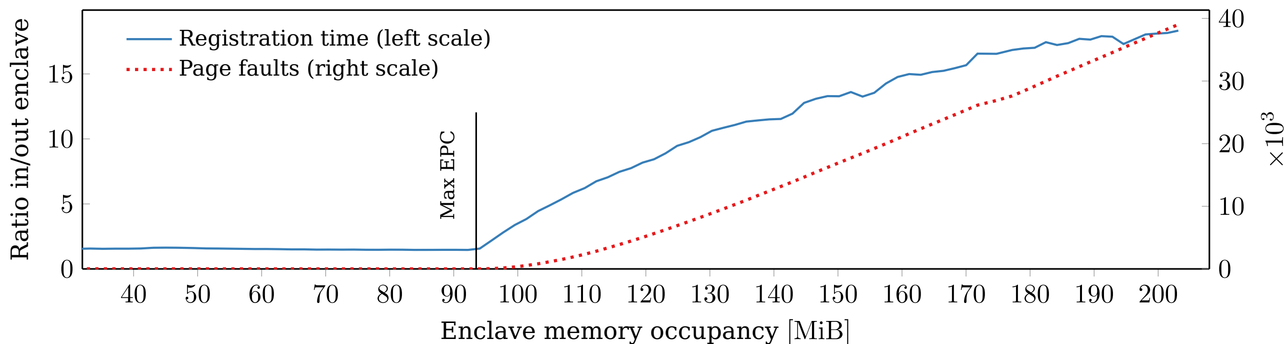

Figure 3.6: Performance loss when surpassing the EPC limit.

Figure 3.6 shows the combined results of two executions when populating the in-memory subscription storage. In one execution we registered subscriptions inside an enclave, and outside in the other. We used the workload *e80a1* in plain-text format, and we executed the same registration code in both experiments. The graph accounts for a moving average of 5,000 points, registering up to 500,000 subscriptions. We plotted the page fault rates by dividing the number of occurrences from the SGX execution over native. The values measured outside are very large for the largest database size, reaching up to 40,000 more page faults.

We also divided the time it took to register one subscription inside the enclave by the time required outside. We can clearly see the point where paging kicks in, when memory consumption reaches just over $93.5\,\mathrm{MiB}$. The vertical line shows the usable EPC memory limit, excluding the reserved memory for SGX internal data structures. At the maximum size of our experiment ($213\,\mathrm{MiB}$), registering a subscription inside the enclave took 18 times more time than doing it outside. These results show that the overhead grows outrageously when paging starts to happen, and they make a strong case for further studies on optimising the memory footprint of applications running inside secure SGX enclaves.

This concludes our evaluation on SCBR. We summarize the results as follows:

- The cost of symmetric encryption is negligible, less than $5\,\mu\mathrm{s}$ of increase in matching time, be it in enclaves or not. In fact, we did not observe any performance difference when using AES in enclave mode in comparison to native execution.
- Exceeding L3 cache level, on the other hand, brings some costs. We observed overheads of 38%, 167% and 164% for the workloads e100a1, e80a1 and e80a4, respectively, with subscription databases of up to $43\,\mathrm{MiB}$.
- The biggest performance issues come when overstepping EPC's limit. In our experiments, up to 18$\times$ for a memory usage of $213\,\mathrm{MiB}$.

3.2 Processing

With the growth of cloud computing, distributed data processing has been extensively studied. Offloading processing to public clouds is a natural choice when organisations do not possess their own data centres, or some specific hardware like graphics processing units (GPUs). Cost cutting can also be one important reason for supporting this decision. Security, however, is still an important concern when it comes to data crunching in third-party infrastructure providers. Even if it is possible to use effective encryption during data transmission and storage, there is no performant way of doing the same when processing it. We have therefore good reasons for enhancing the security in distributed data processing systems. We chose to port a Lua interpreter to run within SGX enclaves and use it as a building block in such systems. Section 3.2.1 describes this port and presents some micro-benchmarks. Next, we use it in two processing frameworks: one that follows the MapReduce programming model for batch processing, described in Section 3.2.2, and

another event stream processing scheme that uses a library inherently asynchronous called RxLua that implements the paradigm of *reactive programming*, described in Section 3.2.3.

In order to provide processing services, there must be a way of receiving the operations that clients want to perform and the data supposed to be processed. Then, these services need to apply the computation on the data and output the results to the next stage in the pipeline. The trusted environment runtime must provide a way to load user instructions, integrate them into the engine and discard them to make room for the following rounds. We discuss here the tradeoffs between different formats of these instructions: binary machine code or *scripts*.

SGX enclave binaries must be previously signed (see Section 2.2.3), so that when interlocutors want to attest it before sharing sensitive information, they are able to know the exact initial state of the protected code, its author and some flags. Based on this, they assess whether to trust it and to continue the data exchange. Although dynamic linking is possible after the enclave creation [66, 174], its use brings some security concerns. In order to execute binary code received after initialisation, the enclave has to allocate memory pages with read, write and execution permissions, or else it would be prevented by the memory management unit (MMU) from running such code. In a way, this weakens the assurance given by the attestation, since code that was not present during initialisation—and therefore not attested—can still be executed. In this case, it is up to the enclave developer to care for the security of dynamically loaded code, which is not an easy task. Moreover, if attackers are able to rewrite these pages by exploring some vulnerability (*e.g.*, stack overflow or control-flow hijacking), they would be able to execute arbitrary code inside the protected environment. Because of such threats, page access rights are signalized in the flags that compose the attestation data, so that an attestor can choose not to interact with enclaves that make use of writable and executable heap.

Interpreted programs are executed by software. Although inherently slower because of that, they are usually more flexible in terms of abstractions and memory management. The code, in this case, is actually input data to the interpreter. Even if the execution of malicious code is still possible, its effects are easier to mitigate. Since there is an intermediary between code and machine, forbidden instructions like system calls or out-of-bounds memory accesses can be previously checked and acted upon. Besides, the machine binary code is entirely available during initialisation, allowing the enclave to have read-only executable pages and therefore inspiring more confidence to whichever process attesting it. For these reasons, we decided to use an interpreted language to operate our distributed processing prototypes.

3.2.1 Lua within enclaves

We settled on a lightweight yet efficient embeddable runtime, based on the Lua virtual machine (VM) [175] and the corresponding multi-paradigm scripting language [176]. Porting legacy code to SGX means that every system call or input/output instruction have to be dealt with, since they are not allowed inside

Table 3.2: Parameters and memory usage for Lua benchmarks.

benchmark name	configuration parameter	memory peak	ratio SGX/Native
dhrystone	5×10^3	275 MiB	1.14
	5×10^6	275 MiB	1.04
fannkuchredux	10	28 MiB	0.99
	11	28 MiB	1.04
nbody	2.5×10^6	38 MiB	0.99
	25×10^6	38 MiB	1.00
richards	10	106 MiB	1.02
	100	191 MiB	0.97
spectralnorm	500	52 MiB	1.00
	5×10^3	404 MiB	0.99
binarytrees	14	25 MiB	1.18
	19	664 MiB	4.76

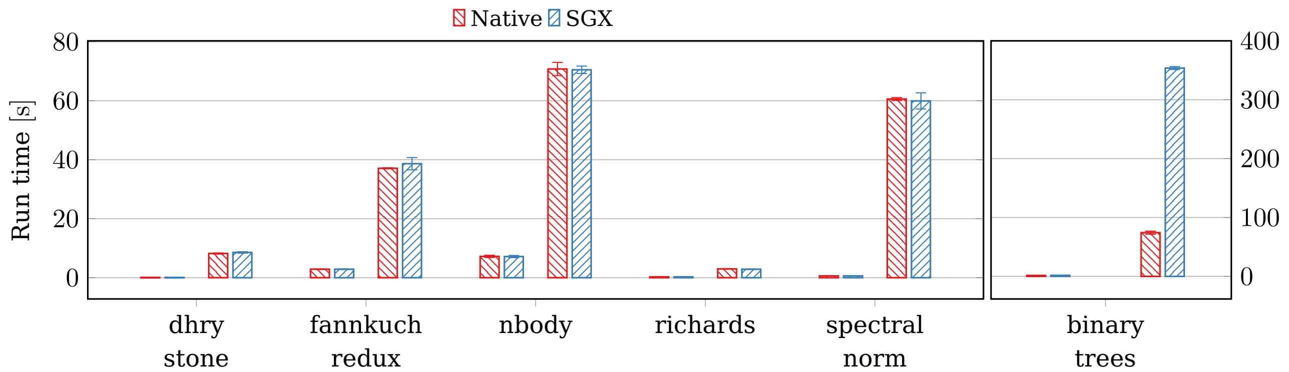

Figure 3.7: Enclave versus native running times for Lua benchmarks.

enclaves. To achieve this, we traced all system calls made by the interpreter to the standard C library and replaced them by alternative implementations that either mimic the real behaviour or discard the call. As a result, attempts made by malicious code to access the hard disk or make network connections are frustrated.

The Lua runtime requires only few kilobytes of memory and is designed to be embeddable. The language API provides the possibility to call Lua functions from C/C++ code. As such, it represents an ideal candidate to execute in the limited space allowed by the EPC. Moreover, application-specific functions can be easily expressed in Lua, including complex algorithms [177]. Our changes to the vanilla Lua source code consist of the addition of about 600 lines of code (LoC), or $2.5\,\%$ of its total size.

We evaluate the raw performances of our SGX Lua interpreter by selecting six available benchmarks from a standard test suite [178]. We based this choice on their library dependencies (by selecting the most standalone ones) and the number of input/output instructions they execute (selecting those with the fewest input and output (I/O)). Each benchmark runs 20 times with the same pair of parameters of the original paper, shown in Table 3.2. Figure 3.7 depicts the total time (average and standard deviation) required to complete the execution of the 6 benchmarks. We use a bar chart plot, where we compare the results of the *Native* and *SGX* modes. For each of the benchmarks, we present two bars next to each other (one per executing mode) to indicate the different configuration parameters used. Finally, for the sake of readability, we use a different y-axis scale for the binarytrees case (from 0 to $400\,\mathrm{s}$), on the right-side of the figure.

In the first version of SGX, it is required to pre-allocate all the memory area to be used by the enclave at initialisation. The most memory-eager test — binarytrees — used more than $600\,\mathrm{MiB}$ of memory. If we compared the running time including the initial allocation, its duration would preponderate for shorter tests. Because of that, we subtracted the allocation time from the measurements of SGX executions, based on the average for the 20 runs. Fluctuations on this event produced slight variations in the execution times, sometimes producing the unexpected result of having SGX executions faster than native ones (by at most $3\,\%$). Table 3.2 lists the parameters along with the maximum amount of memory used and the ratio between run times of SGX and Native executions. When the memory usage is low, the ratio between the Native and SGX versions is small, less than $20\,\%$ in our experiments. However, when the amount of memory usage increases, performance drops to almost $5\times$ worse, as in the case of the *binarytrees* experiment. As we already observed in previous sections, the smaller the memory usage, the better performance we can obtain from SGX enclaves when compared to native executions.

3.2.2 Lightweight MapReduce

Since its adoption by Google [179], the MapReduce programming model consistently gained ground as a viable solution for assuring the necessary scalability of distributed data processing. The generic model, composed of the *map* and *reduce* functions, was widely used to implement applications that can leverage parallel task processing. The data to be processed are made available to a set of *mapper* nodes, which

apply in parallel a *map* function responsible for converting individual data items to a finite set of key and value pairs. The output is redistributed based on the keys, so that all values for a given key is grouped in one single *reducer* node, in a step called *shuffle*. Reducer nodes, in turn, execute in parallel a *reduce* function for processing each dataset and outputting a final result. This model can be used in processing tasks that range from simple compoundable operations like counting, sorting, and searching data; to more complex algorithms like cross-correlation or page rank. MapReduce was adapted in different ways to fit a wide diversity of scenarios and deployment platforms (see Section 2.4.2).

We propose a self-contained framework for securing MapReduce that leverages SGX [12]. Our system combines SCBR for communication (see Section 3.1.1), a Lua interpreter as processing engine (see Section 3.2.1), and a MapReduce library. It is independent of the particular characteristics of the *map* and *reduce* functions and can hence be used for any problem that can be made parallel with MapReduce. The specific code to be executed in the MapReduce service can be integrated in simple scripts, which run in isolation using SGX over data decrypted only inside the enclave. Our focus is on providing a flexible framework for securely running MapReduce applications that can be easily implemented and deployed. The basic *word count* MapReduce example, for counting the number of occurrences of different words in a given text, can be implemented in our framework with less than 30 LoC. We evaluate our approach using the widely used k-means clustering algorithm, showing that the overhead incurred is minimal and that our solution is applicable to other use cases.

Solution architecture

Figure 3.8 displays the entities composing our solution: clients, SCBR pub/sub engine and workers, which can assume the role either of a mapper or a reducer. Clients provide the code to be executed, the data to be processed and gather the results after completion. In our work, we are concerned with assuring the privacy and integrity of code and data processed within the *map* and *reduce* functions, while offering to the programmer an accessible lightweight environment of implementing various use-cases. Data or code is only seen in plain-text form at the client premises or inside enclaves. For simplicity, we assume a shared key SK was previously established between clients and workers.

All communication channels use the ZeroMQ [180] message passing library, having a central point in the SCBR engine. Although such a centralised approach is not suitable to large-scale data processing, it is arguably useful for modest quantities of highly sensitive data that could be, for instance, partitioned from higher amounts of non-sensitive data. Nevertheless, it has been demonstrated [170, 181] that it is possible to elastically scale a pub/sub engine by specializing its functional steps into replicable operators. That could dramatically improve the network performance of such a centralised approach.

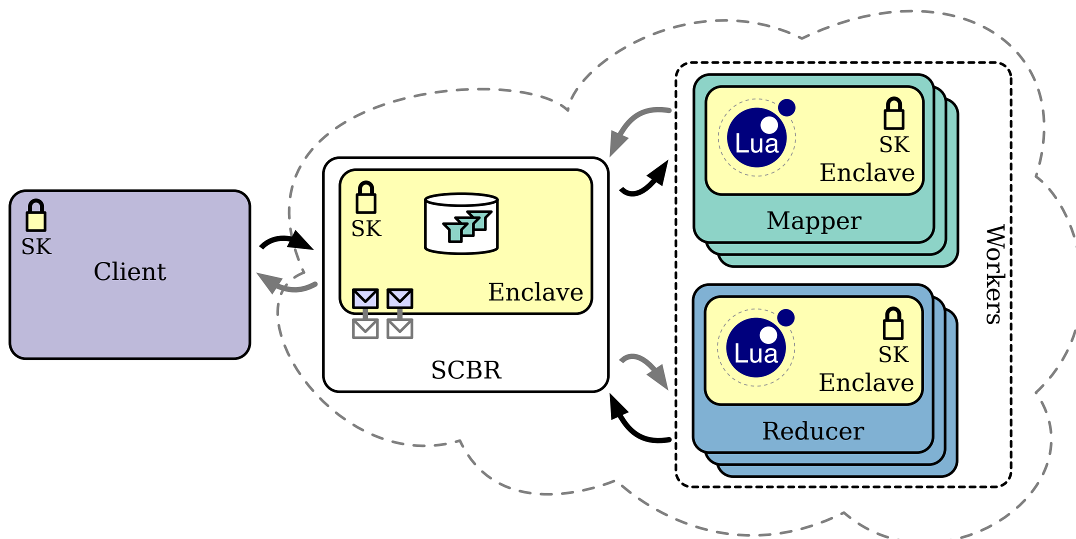

Figure 3.8: Lightweight MapReduce components.

Batch processing is the act of transforming one large and finite group of data that is entirely available upfront. It contrasts with the processing of events, which are smaller pieces of data that are individually processed and timely unbounded. In our system, a batch process is bootstrapped with an initial protocol among all parties and with the provisioning of code and data made by the client, when he sends chunks of the input data to mappers. We describe bellow what happens next.

1. Mapper nodes independently execute a *map* function on their input data chunk and output a collection of n key and value pairs $\{< k_0, v_0 >, < k_1, v_1 >, ..., < k_{n-1}, v_{n-1} >\}$. Optionally, pairs with common keys can be aggregated by a combiner function in order to optimize the redistribution, *e.g.*, if $k_0 = k_1$ then a *combine* function may be called with $< k_0, \{v_0, v_1\} >$ as input. Conceptually, *combine* acts like a local *reduce* function, in the context of a single mapper.

2. $< k, v >$ tuples are shuffled according to their keys and redistributed to reducer nodes. All tuples corresponding to the same key arrive on the same reducer.

   For instance, if the outputs of three mappers m_i are:

   - $m_0 \rightarrow \{< k_0, v_0 >, < k_1, v_1 >\}$
   - $m_1 \rightarrow \{< k_1, v_2 >, < k_2, v_3 >\}$
   - $m_2 \rightarrow \{< k_0, v_4 >, < k_2, v_5 >\}$

   As a result of the shuffling phase, three reducers r_i could receive as input:

   - $r_0 \leftarrow < k_0, \{v_0, v_4\} >$
   - $r_1 \leftarrow < k_1, \{v_1, v_2\} >$
   - $r_2 \leftarrow < k_2, \{v_3, v_5\} >$

3. Reducer nodes independently execute the *reduce* function on all data associated to a given key and output the result of their computation, which is forwarded back to the client.

MapReduce as topic-based pub/sub

The SCBR engine is responsible for securely storing subscriptions that contain the conditions under which each message is forwarded to the corresponding interested party. To design the session establishment protocol, we first list in Table 3.3 the interests of each role, *i.e.*, what data kind each entity is interested in. This will later define their subscriptions.

We decided to map payload data into topics [83]. By tagging each message with the corresponding content, subscriptions and publications are respectively made according to the columns *Input* and *Output* in Table 3.3. Besides these messages, we also identified the need of having two more topics, so that idle workers (mappers and reducers) are notified when there is a new client willing to process a batch of data (job advertisement). Conversely, clients need to know when a worker is willing to serve them (apply for position). Table 3.4 lists all topics. *Workers* stands for both mappers and reducers.

Worker nodes will act as subscribers registering queries to find out about new MapReduce job openings, and also as publishers to signal their availability and type (mapper or reducer) in a job application. The

Table 3.3: MapReduce: Payload used as inputs and outputs for each role.

Role	Input	Output
Client	Final Results	Map code Reduce code Mappable data
Mapper	Map code Mappable data	Reducible data
Reducer	Reduce code Reducible data	Final results

Table 3.4: MapReduce: Topics for the pub/sub based protocol.

Description	Symbol	Subscribers	Publishers
Map code	MAP_CODETYPE	Mappers	Clients
Reduce code	REDUCE_CODETYPE	Reducers	Mappers
Mappable data	MAP_DATATYPE	Mappers	Clients
Reducible data	REDUCE_DATATYPE	Reducers	Mappers
Final results	RESULT_DATATYPE	Clients	Reducers
Job advertisement	JOB_ADVERTISE	Workers	Clients
Apply for position	JOB_APPLY	Clients	Workers

client of the MapReduce service, which is also the data owner, will both act as subscriber and publisher, registering subscriptions for job applications and publishing advertisements on new MapReduce jobs to be executed. Moreover, the client will publish code and data to the registered workers and obtain the results after job completion.

The MapReduce processing starts with an initial message exchange, as shown in Figure 3.9. For simplicity, we use a reducer node as worker, but an analogous procedure applies for mappers. Worker nodes register their intent of being notified for MapReduce job openings through subscriptions on the topic JOB_ADVERTISE. The client registers its interest in knowing when workers are ready to perform a task by subscribing to the topic JOB_APPLY. When the client has a new job to execute, it advertises it through a JOB_ADVERTISE publication, which is received by workers that previously registered for this topic. Idle workers notify then their readiness to execute the advertised job through a JOB_APPLY publication. They also include in the message payload their subscriptions for code and data (particular to the role they choose: mapper or reducer). At the end of this negotiation, if the client decides to hire a worker, it registers on SCBR the received subscriptions for code and data on the worker's behalf. By doing so, the client establishes the MapReduce chain and keeps track of how many workers it has hired and of which kind (mappers and reducers).

The provisioning of code and data is shown in Figure 3.10. Besides the code itself, the client includes the number of reducers along with the Lua scripts in case of MAP_CODETYPE publication topic, or the number of mappers in case of type REDUCE_CODETYPE. The purpose is to make the workers aware of how many messages indicating the stream's termination that they have to wait before considering the work done. This is important because the reduce phase can only start once all data for a given key is routed to the

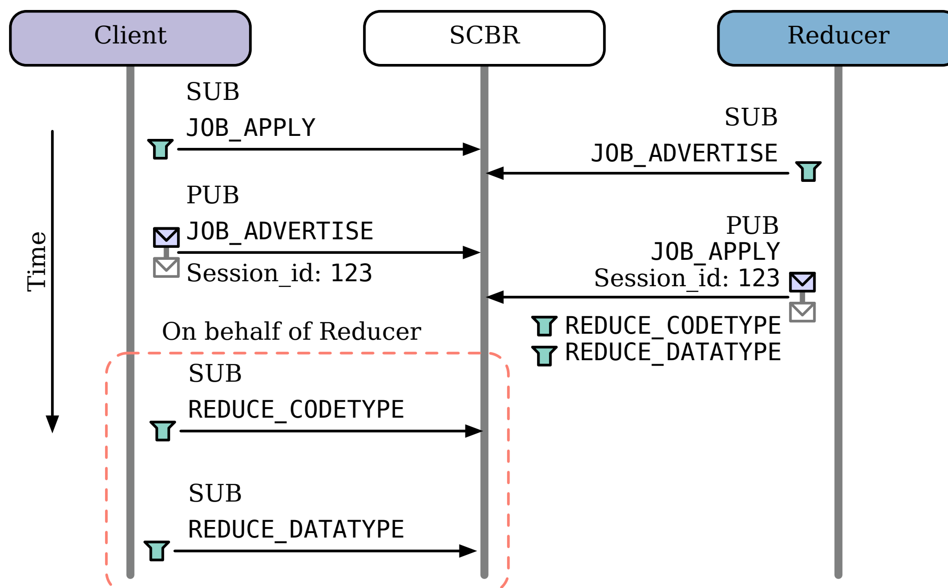

Figure 3.9: MapReduce: Session establishment protocol.

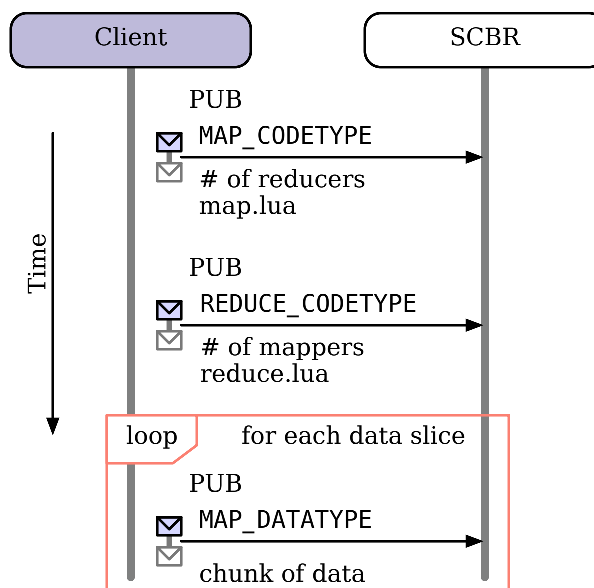

Figure 3.10: MapReduce: Provisioning of code and data.

intended worker. Additionally, the amount of reducers (`rcount` in Listing 3.1) received by a mapper is used in a hash function that takes as argument the tuple key and returns the indication of which reducer it has to be forwarded to. After sending the code, data is split by the client among the mappers. The destination identifier is included in the header of the `MAP_DATATYPE` publication.

Workers decrypt the received code and store it inside the enclaves. When data arrive, mappers perform the processing and each one of the resulting key-value pairs is forwarded to the proper reducer. The reducer's identifier is obtained after providing the key and number of reducers as arguments to the hash function that comes along with the code of the mapper. The shuffling phase is hence conducted by the mappers. In order to forward data to the following step, all that the Lua script has to do is calling a special function called *push(key,value)*, and the framework handles all the communication aspects of forwarding the data.

Listing 3.1 shows the sample code of a mapper of a word count application. The script can contain as many helper functions as desired. The following special functions, however, are called by the framework:

```
function hash(key,rcount)
    return string.byte(key,1) % rcount
end

function combine(key,values)
    local sum = 0
    for k,v in pairs(values) do
        sum = sum + v
    end
    push(key,sum)
end

function map(key,value)
    for word in value:gmatch("%w+") do
        push(word,1)
    end
end
```

Listing 3.1: Map code in Lua for word count.

```
function reduce(key, values)
    local sum = 0
    for k,v in pairs(values) do
        sum = sum + v
    end
    push(key,sum)
end
```

Listing 3.2: Reduce code in Lua for word count.

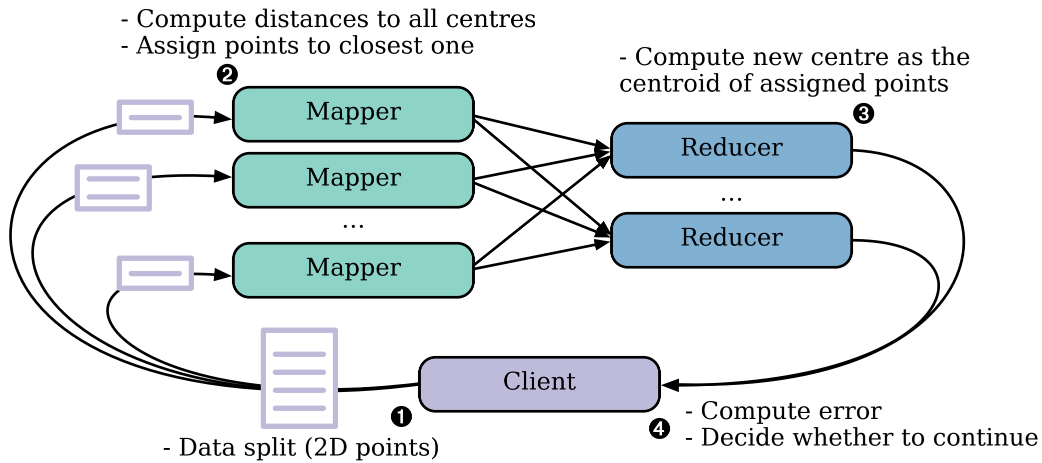

Figure 3.11: K-means with MapReduce.

map(key, value)	Contains the functional implementation of mapper.
combine(key, values)	[Optional] Post-processing on values grouped by keys.
hash(key, rcount)	Returns the reducer id that is supposed to receive a given key considering that there are rcount reducers in total.

Likewise, listing 3.2 shows a sample code for the reduce step that contains a single special function:

reduce(key, values)	Contains the functional implementation of reducer.

Evaluation

To test our solution, we use the k-means clustering method for data classification. K-means [182] is an unsupervised learning algorithm (*i.e.*, it does not depend on training data) widely used for data classification. It operates as following:

(i) A certain k number of clusters is fixed *a priori* and k corresponding centres are defined for each one of them;

(ii) Data items are iterated and assigned to the nearest cluster centre (typically through Euclidean distance); and

(iii) Every cluster centre is recomputed as the centroid of the assigned data items (the mean of those points).

Steps (ii) and (iii) repeat until a termination criterion is reached (*e.g.*, the sum of distances between the old and the new cluster centres is below a given threshold). For implementing k-means in the MapReduce model we put (ii) in the *map* function and (iii) in the *reduce* function as displayed in Figure 3.11. The termination criterion is checked by the client, who decides whether to iterate again. If it does, the centres input for mappers are replaced by the most recently calculated ones.

We conducted all the experiments using two SGX-capable machines, both with processor Intel i7-6700 64bits, clock of 3.4GHz, 8MB cache, 4 cores, 8 threads, with 8GB of installed memory and solid-state drive (SSD) of 256GB. In terms of software, we used the Intel SGX software development kit (SDK) 1.7.100 over Ubuntu 14.04.1, kernel 4.2.0-42. Unless mentioned otherwise, messages were all encrypted with AES-CTR [183] with key and input vector both of 128 bits and were decrypted only inside the enclaves. The process placement was made as follows: Machine 1: the client, 8 mappers and 5 reducers. Machine 2: 8 mappers and 5 reducers. The number of mappers was chosen to be twice as much as the number of cores in each machine to take advantage of parallelism, while the number of reducers was set to be a divisor of input data size to stimulate an even distribution of work among them. To illustrate how small our final code-base is, Table 3.5 shows the memory section sizes of executables and shared libraries that are loaded into enclaves.

Table 3.5: Lightweight MapReduce: binaries' size.

	text	data	bss	sum
client	371 KiB	26 KiB	376 B	397 KiB
worker	282 KiB	26 KiB	768 B	308 KiB
worker enclave	663 KiB	59 KiB	82 KiB	803 KiB
scbr	271 KiB	14 KiB	72 B	285 KiB
scbr enclave	272 KiB	7 KiB	79 KiB	358 KiB

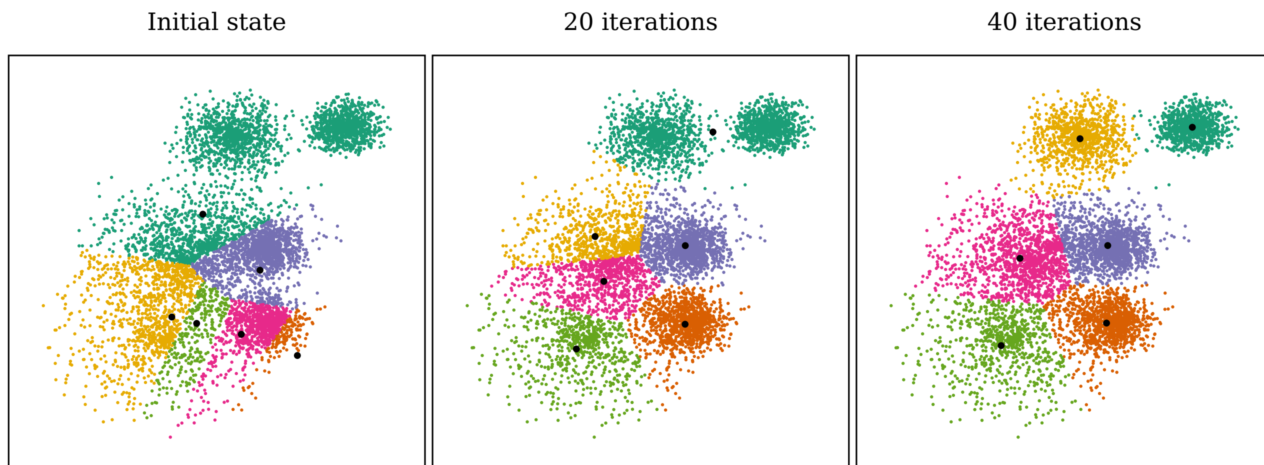

Figure 3.12: K-means example with 6 centres and 7000 data points.

Figure 3.12 illustrates 3 out of 40 iterations that k-means took to converge. In this example, we synthetically generated 7,000 observation points and 6 centres. As it can be seen in the first frame, the initial centroids are far from an even distribution across the grid. After 20 iterations, the centroids assume closer positions to those of the final result, which is achieved at the 40th iteration. We arbitrarily set the threshold to be one thousandth of the diagonal of the rectangle that contains all the observed points. That means that the iterative process finishes when the average distance of centroids between two subsequent iterations is less than that fraction. In such algorithm, the final result and convergence speed depend on the initial points. Besides, there is no guarantee that the solution is the global optimum.

Next, we conducted experiments to assess the influence of input data sizes, *i.e.*, the number of observation points n and centroids k, on the memory usage and processing time. Figure 3.13 shows the average time it took to complete one iteration of Kmeans with varying input sizes. It can be noticed that, although the variation on the number of clusters can cause some inflection in the curves, the completion time is mostly affected by the number of observed data points. Moreover, while the two first increments on the number of data points ($n = 10k$ and $n = 100k$) caused a proportional increase on consumed time regarding the data growth (ten times), the last one ($n = 1M$) induced a twenty-fold rise. That can be explained by the growth in the occurrence of cache misses within each worker. When that happens, data must be fetched from main memory. When using SGX protected executions, this means one page has to be evicted from cache (and hence, encrypted), while the one that is fetched must be decrypted and checked for integrity and freshness.

To better analyse these cache effects, we decided to measure cache miss rates. Since the *reduce* phase is more memory intensive, we chose to make the average on the cache miss rates per second of all 10 reducers in each execution, *i.e.*, for the same second, cache miss rates of reducers were summed and divided by 10. Wall clocks were synchronized with a common NTP server, so that the resulting skew was at the range of tens of milliseconds and should not affect the sampling resolution of $1\,\mathrm{s}$. Figure 3.14 shows that measurement as reported by the tool `pidstat` when the number of centroids is $k = 50$. Note that *y* scale is logarithmic, so that the cache miss rates for $n = 1M$ is at least two orders of magnitude higher than $n = 100k$. Valleys in the curves represent the interval between iterations, which is more apparent in the larger experiments.

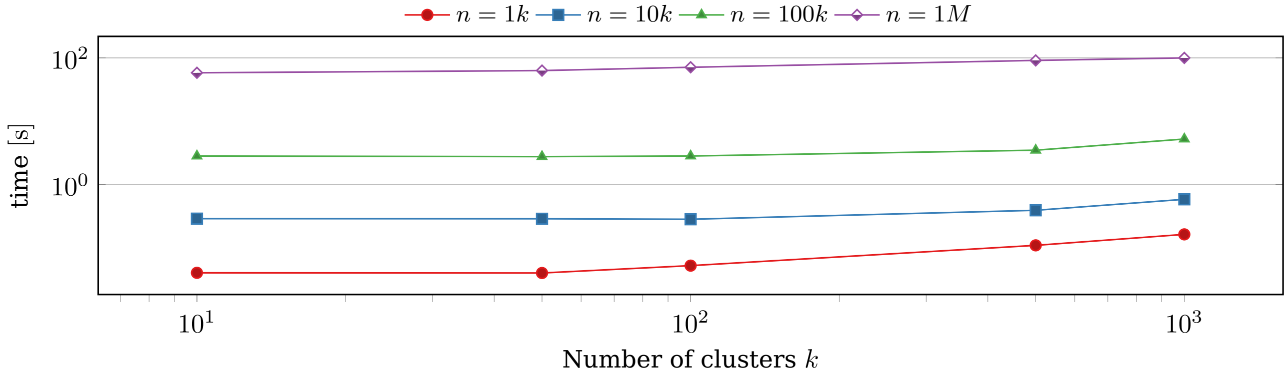

Figure 3.13: MapReduce: Average time to run one iteration with varying input sizes.

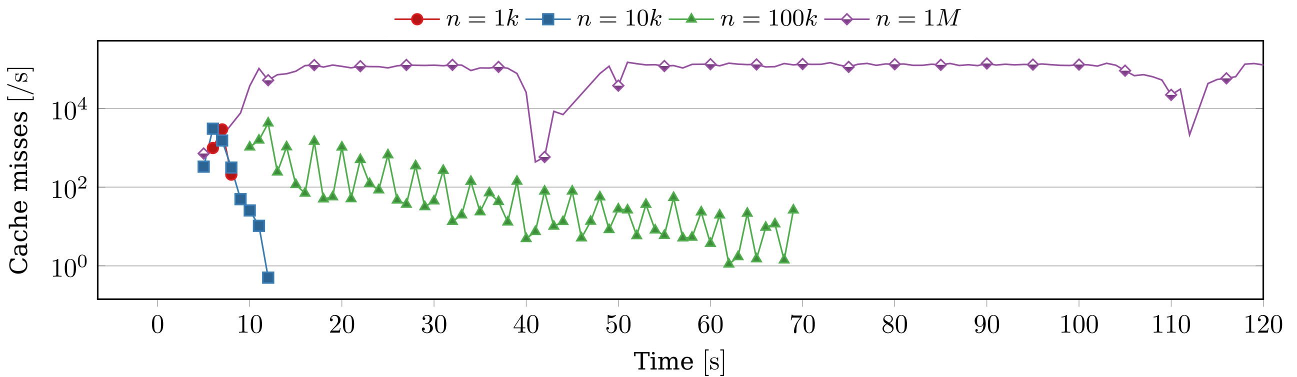

Figure 3.14: MapReduce: Cache miss rates for different input sizes.

Finally, we measured the influence of SGX when compared to native executions, *i.e.*, with no hardware protection. We ran the same datasets with a fixed number of clusters $k = 50$ and varying the number of observed points from $n = 1000$ until $n = 1M$. Results are plotted in Figure 3.15. Table 3.6 shows the data volume exchanged in each MapReduce step for these experiments. We also include the ratio between SGX and native run times. The time corresponds to the average of all iterations in a single run of k-means (until the threshold was reached). Coefficient of variation across multiple runs was negligible. Enclave execution overhead is kept around 35% until we start to get high occurrence of cache misses, as discussed before, when it reaches more than 200%.

This concludes our experiments with lightweight MapReduce. As in Section 3.1.1, we notice again some expressive overhead after using more than the processor's cache limit. Based on its size, we can establish the maximum amount of data that a single SGX-capable machine would be able to handle before incurring in too much overhead. In our experiments, we perceived that behaviour when processing amounts somewhere in between 11 MiB and 96.5 MiB shared between two machines (average of around 54 MiB, or 27 MiB per machine). A rough estimation based on our empirical evaluations would be to limit those amounts to three times the cache size, or 24 MiB in our case. Scalability could be achieved horizontally, with the

Table 3.6: MapReduce: Data volume exchanged per iteration.

	Split	Shuffle	Output
$n = 1k$	58.7 KiB	112.1 KiB	4.3 KiB
$n = 10k$	257.2 KiB	1.1 MiB	4.5 KiB
$n = 100k$	2.2 MiB	11 MiB	4.6 KiB
$n = 1M$	19.1 MiB	96.5 MiB	4.6 KiB

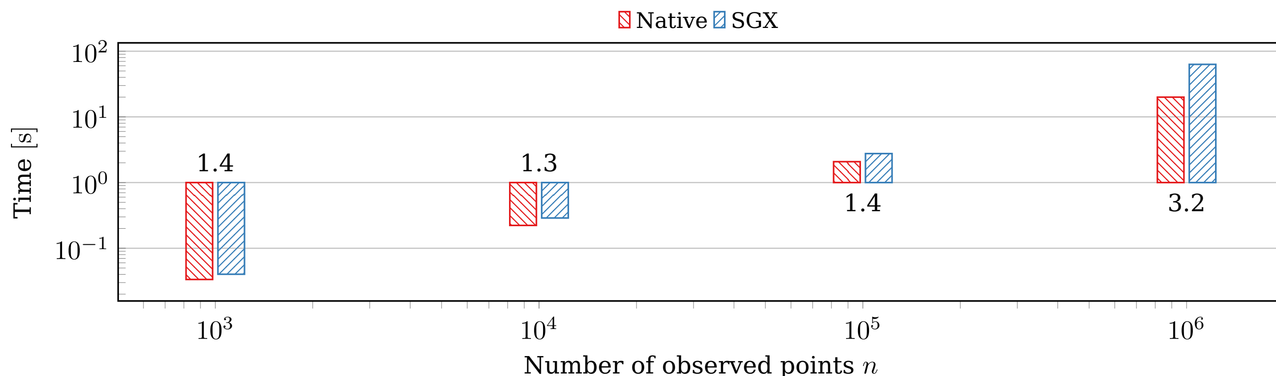

Figure 3.15: MapReduce: SGX overhead.

addition of more machines. This reinforces the lightweight aspect of our approach, both in terms of the framework code size (2 MiB summing up all components) and its capacity before incurring into important overheads (24 MiB per machine in the shuffling phase).

3.2.3 SecureStreams

The Internet of things (IoT) has fostered the emergence of novel data analytics and processing technologies to support the continuous flow of information gathered by a large amount of sensing devices. Since these data streams may convey sensitive information, stream processing requires support for end-to-end security guarantees in order to prevent third parties from accessing restricted data. We present SecureStreams [14], a middleware framework for developing and deploying secure stream processing on untrusted distributed environments.

SecureStreams supports the implementation, deployment, and execution of stream processing tasks in distributed settings. It employs a message-oriented middleware [184], the transport layer security (TLS) protocol [29] for communication and SGX to deliver end-to-end security guarantees along data stream processing pipelines. SecureStreams can scale vertically and horizontally by adding or removing processing nodes at any stage of the pipeline. Its design is inspired by the dataflow programming paradigm [185], where the developer combines independent processing components (*e.g.*, mappers, reducers, sinks, shufflers, joiners) to compose specific processing pipes. Regarding packaging and deployment, SecureStreams smoothly integrates with a lightweight virtualisation technology, namely Docker [186].

Architecture

SecureStreams combines two base components: *worker* and *router*. A worker continuously listens for incoming data by means of non-blocking I/O. As soon as data flows in, an application-dependent business logic is applied. Router components act as message brokers between workers according to a given *dispatching policy*. A typical use-case is the filter/map/reduce pattern from the functional programming paradigm [187], in which each worker executes one of these functions. Figure 3.16 depicts a possible implementation of this dataflow using the SecureStreams middleware.

SecureStreams is designed to support the processing of sensitive data inside SGX enclaves. Each component is wrapped inside a lightweight Linux container (in our case, the industrial standard Docker [186]). Each container embeds all the required dependencies, while guaranteeing the correctness of their configuration, within an isolated and reproducible execution environment. By doing so, a SecureStreams processing pipeline can be easily deployed without changing the source code on distinct public or private infrastructures. Containers' deployment can be transparently executed on a single machine or a cluster, using a Docker network and the Docker Swarm scheduler [188].

Like in SCBR (Section 3.1.1), communication is done with ZeroMQ, a high-performance asynchronous messaging library [172]. Each router component hosts inbound and outbound queues, according to ZeroMQ's

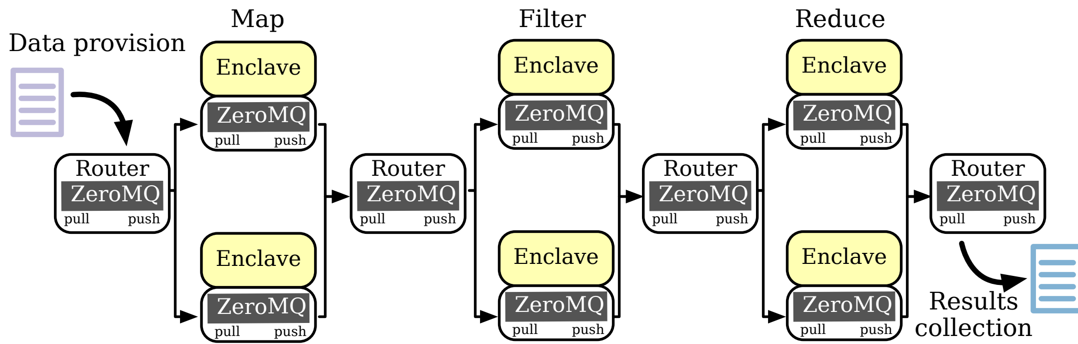

Figure 3.16: Example of SecureStreams pipeline.

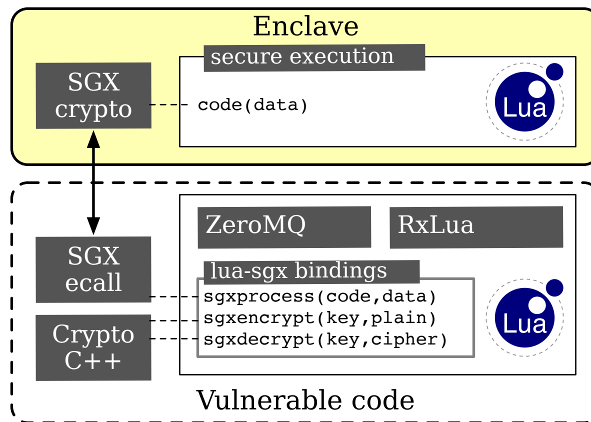

Figure 3.17: Integration between Lua and Intel SGX.

pipeline pattern [189]. Messages are streamed from a set of *push* peers, *i.e.*, the upstream workers in the pipeline. They use *push* sockets that send messages to downstream workers in a round-robin fashion. These, in turn, have an inbound pull socket, which uses fair-queuing scheduling to deliver messages to upper layers.

We define the processing pipeline components and their chaining by means of Docker's Compose [190] description language. Once the processing pipeline is defined, the containers can be deployed on the computing infrastructure. We use the `constraint` placement mechanism to make the Docker Swarm's scheduler deploy workers requiring SGX on appropriate hosts.

Implementation

SecureStreams is implemented in Lua version 5.3. The implementation of the middleware itself requires careful engineering, especially with respect to the integration with SGX enclaves. A SecureStreams application, on the other hand, can be implemented in remarkably few lines of code. For instance, the implementation of the map/filter/reduce pipeline accounts for only 120 LoC (not including dependencies). The framework partially extends RxLua [191], a library for reactive programming in Lua. RxLua provides the required API to design a data stream processing pipeline following the dataflow programming pattern [185].

Listing 3.3 provides an example of a RxLua program (and consequently a SecureStreams program) to compute the average age of a population by chaining `:map`, `:filter`, and `:reduce` functions. The `:subscribe` function performs the subscription of 3 functions to the data stream. Following the *observer* design pattern [192], these functions are observers, while the data stream is an observable.

SecureStreams ships the business logic for each component into a dedicated Docker container and executes it. Communication between routers and workers happens through ZeroMQ (version 4.1.2) and the corresponding Lua bindings [193]. The framework safely forwards data and code to enclaves, so that they

```
Rx.Observable.fromTable(people)
 :map(
   function(person)
     return person.age
   end
 )
 :filter(
   function(age)
     return age > 18
   end
 )
 :reduce(
   function(accumulator, age)
     accumulator[count] = (accumulator.count or 0) + 1
     accumulator[sum] = (accumulator.sum or 0) + age
     return accumulator
   end
 )
 :subscribe(
   function(data)
     print("Adult␣people␣average:", data.sum / data.count)
   end,
   function(err)
     print(err)
   end,
   function()
     print("Process␣complete!")
   end
 )
```

Listing 3.3: Example of process pipeline with RxLua.

do not have to operate on files or any other I/O. In case such attempts occur, they are frustrated by sanitized versions of *libc* procedures. By adopting this set of measures, SecureStreams safely abstracts the underlying network and computing infrastructure from the developer perspective.

We reuse here the SGX Lua interpreter described in Section 3.2.1. Additionally, we include in the enclave both *json* [194] and *csv* [195] parsers to ease the development of SecureStreams applications. With these libraries, the enclave size containing the complete runtime remains small, approximately 220 KiB (19 % larger than the original). Besides the SGX Lua interpreter, we still had to provide support for communication and the reactive streams framework itself. For this, we use an external vanilla Lua interpreter, with a couple adaptations to allow the interaction with the enclaves and the Lua interpreter therein. Figure 3.17 shows the resulting scheme. We extend the Lua interface with 3 functions: `sgxprocess`, `sgxencrypt`, and `sgxdecrypt`. The first one forwards the encrypted code and data to be processed in the enclave, while the remaining two provide cryptographic functionalities. We assume that attestation and key establishment was previously performed.

Evaluation

We conducted our experiments on 2 machines using the Intel Core i7-6700 processor [196] and 8 GiB random-access memory (RAM) running Ubuntu 14.04.1 long term support (LTS) (kernel 4.2.0-42-generic). Each node runs Docker (v1.13.0) and joins a Docker Swarm [188] (v1.2.5) using the Consul [197] (v0.5.2) discovery service. The Swarm manager and the discovery service are deployed in a third machine. Containers that compose the pipeline leverage the Docker overlay network to communicate with each other. Machines are physically interconnected using a switched 1 Gbps network.

To evaluate our system, we chose a dataset released by the *American Bureau of Transportation Statistics* [198]. It reports on flight departures and arrivals of 20 air carriers [199]. We implemented a SecureStreams application to compute average delays and the amount of delayed flights of each air carrier (inspired by [200]). Table 3.7 specifies the code size of each application component. We implemented a processing pipeline that

Table 3.7: Code size of a SecureStreams application.

System layer	Size (LoC)
DelayedFlights app	86
SecureStreams library	350
RxLua runtime	1,481
Total	1,917

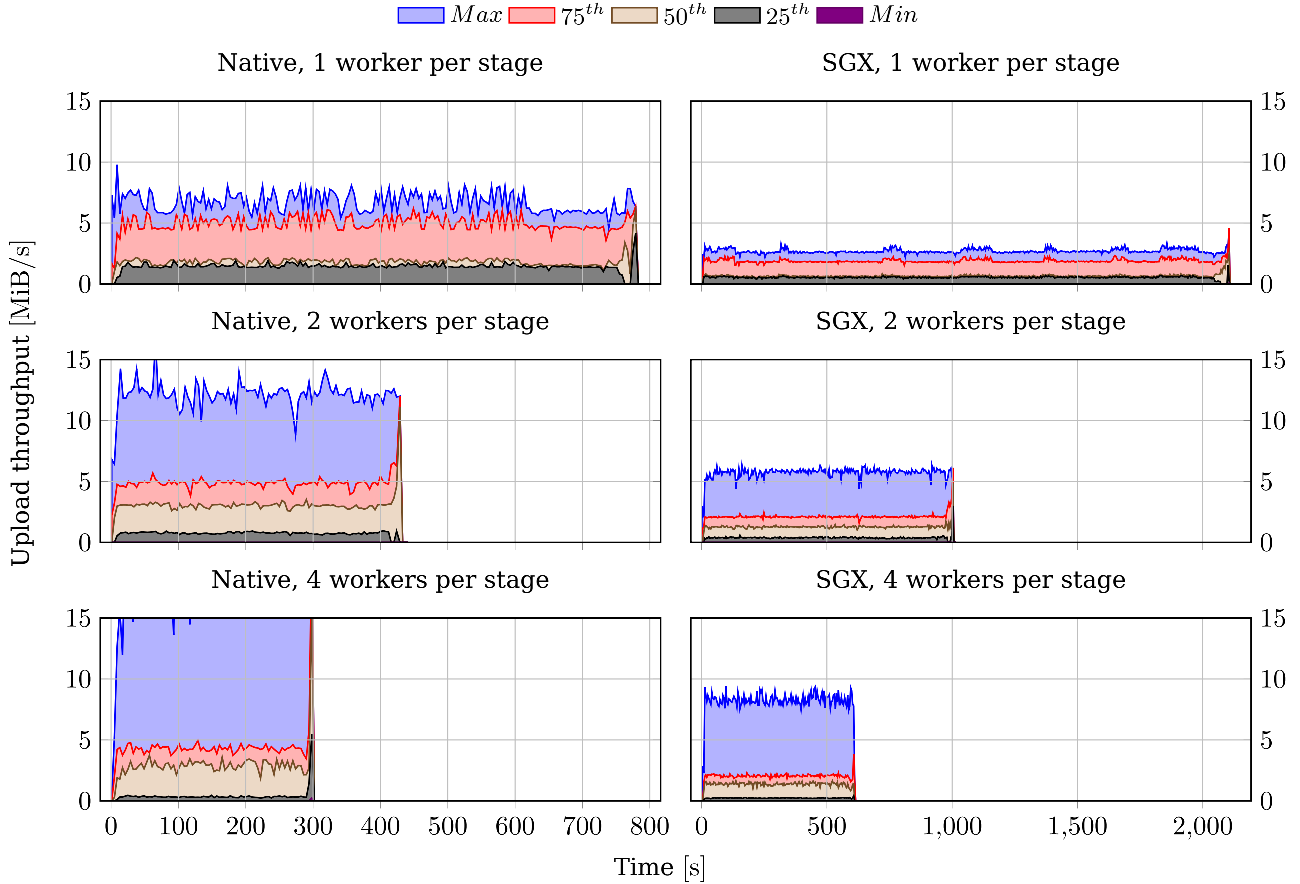

Figure 3.18: SecureStreams: Throughput comparison between native and SGX versions with 1 and 4 workers per stage.

(i) *maps* the input dataset, in comma separated value (CSV) format, into a data structure;

(ii) *filters* out irrelevant data, *i.e.*, only let data concerning delayed flights go through; and

(iii) *reduces* the filtered data by computing the output information.

We use the 4 last years of the available dataset (from 2005 to 2008), for a total of 28 million entries and $2.73\,\mathrm{GiB}$ of data. To measure the achievable throughput and the network overhead of our system, we deploy the SecureStreams pipeline in 2 configurations:

1. The baseline does not use enclaves. The input dataset is encrypted before its injection in the pipeline, so that data transmission is secure. This approach is however unsafe for deployment in untrusted infrastructures since data is decrypted before being processed.

2. In the SGX deployment, the input dataset is encrypted and the data processing is operated inside enclaves.

Data nodes inject the input dataset as fast as possible while bandwidth measurements are collected from Docker's internal monitoring and statistical module. For each configuration, we vary the number of work-

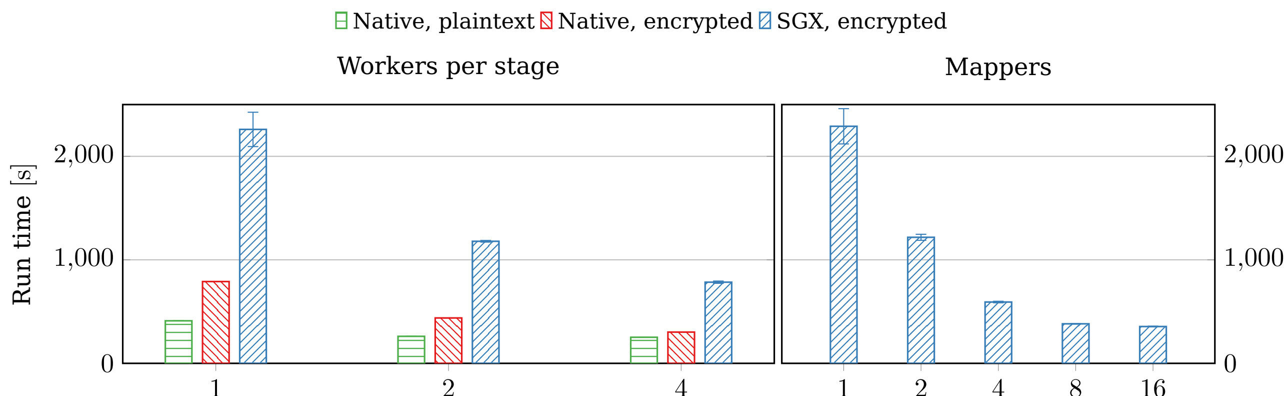

Figure 3.19: Average processing time when varying number of workers.

ers per stage, from one to four. Figure 3.18 shows the results in the form of stacked percentiles. To exemplify, the median (50th percentile) throughput at $200\,\mathrm{s}$ when operating with four nodes per stage corresponds to $3.1\,\mathrm{MiB/s}$ and $1.5\,\mathrm{MiB/s}$ for native and SGX executions, respectively. This means that 50% of the nodes output data at no more than these rates at that moment in time.

The baseline configuration, *i.e.*, native execution with 1 worker per stage, completes in $794\,\mathrm{s}$ with median of $1.8\,\mathrm{MiB/s}$ and peak of $9.8\,\mathrm{MiB/s}$. Doubling the number of workers reduces the processing time down to $442\,\mathrm{s}$, a speed-up of $1.8\times$ (median: $3\,\mathrm{MiB/s}$, peak: $16.5\,\mathrm{MiB/s}$). Scaling up the workers to 4 per stage in the native execution makes the dataset be consumed in $302\,\mathrm{s}$ (median: $2.9\,\mathrm{MiB/s}$, peak: $27\,\mathrm{MiB/s}$[1]), or $1.5\times$ speed-up considering the previous experiment.

When using SGX enclaves, data processing slows down by $2.7\times$ ($2120\,\mathrm{s}$) for the setup with one worker per stage. We also pay a penalty in terms of overall throughput—*i.e.*, the median rarely exceeds $650\,\mathrm{KiB/s}$, peak of $4.6\,\mathrm{MiB/s}$. Increasing the workers per stage renders speed-ups of $2.1\times$ (median: $1.3\,\mathrm{MiB/s}$, peak: $5.9\,\mathrm{MiB/s}$) and $1.6\times$ (median: $1.4\,\mathrm{MiB/s}$, peak: $9.9\,\mathrm{MiB/s}$), with 2 and 4 workers, respectively. In comparison to native execution though, they present slow-downs of $2.3\times$ and $2.1\times$.

We further evaluate SecureStreams in terms of scalability by establishing correlations among running time, number of worker nodes and available hardware resources. Specifically, the number of processing units. We do so by changing the number of workers, like in the previous experiment, and also by only varying the number of mappers, the most significant stage in terms of overall performance. Additionally, as baseline, we experiment with plain-text data across the whole pipeline. We repeat each configuration for 5 times. Figure 3.19 shows the mean runtime, with error bars corresponding to the standard deviation.

First, we increase the number of workers in each stage of the pipeline. We observe an ideal acceleration when going from the configuration using 1 worker per stage to 2. In the setup using 4 workers, on the other hand, we do not reach the same speed-up. This mainly happens due to the number of deployed containers, which becomes greater than the amount of available cores in each processor. Containers account for the sum s of input data streams i, routers $r = 4$ and workers. The amount of the latter is equal to the number of stages in the pipeline (3 in our setup, see Figure 3.16) multiplied by the number of workers per stage w. Besides, we define the number of input data streams as $i = w$. The total number of containers is therefore $s = 4w + 4$, or $s = 20$ when $w = 4$, which is greater than the number of physical processing units in our setup (2 machines with 8 cores each, *i.e.*, 16 cores).

Finally, we observe the running time when varying from 1 to 16 the number of mappers in the first stage of the pipeline, as it is the most computational intensive one number of filters and reducers is kept constant. Again, we observe an ideal speed-up ($2\times$) until the number of deployed containers reaches the amount of physical cores. Beyond that, the improvement is only marginal ($1.07\times$ from 8 to 16 mappers). These

[1]not visible in the figure due to visualisation purposes, so that y-axis scale is kept consistent for comparison while allowing some noticeable detail in lower throughput experiments

experiments show that SecureStreams' scalability is primarily limited by the total number of physical cores available, for a map/filter/reduce pipeline.

Apart from that, other factors contribute for limited streaming throughput. We observed, for instance, that the system does not saturate the available network bandwidth. We believe this behaviour can be explained by the lack of optimisations in the application logic and tuning options of the inner ZeroMQ queues. For this reason, we did not reach memory occupancy beyond the EPC limit, which would dramatically deteriorate performance (see Section 3.1.1). Nevertheless, we reached our primary goal of finding out the overhead when using SGX enclaves for processing real data in a streaming pipeline, along with a couple insights about scalability. As expected, one should anticipate longer processing times (roughly $2-3\times$ in our experiments) and lower throughputs when executing stream processing within SGX enclaves. Disadvantages of this trade-off between security and runtime can be mitigated by parallelisation, as long as the workload is adapted to hardware resources, as our experiments indicate.

3.3 Summary

In this chapter, we proposed and empirically evaluated a set of distributed systems that use TEEs and naturally fit in cloud scenarios. They are therefore supposed to be deployed in third-party administrative domains, which are commonly reputed as hostile and unsafe environments.

First, we presented the architecture and evaluation of SCBR (Section 3.1.1), a secure content-based routing engine that takes advantage of SGX enclaves. In doing so, we were able to leverage state-of-the-art techniques for efficient filtering in plaintext, since the trusted perimeter is limited to the CPU die. Outside that boundary, private data in main memory is always encrypted and protected from tampering and replay attacks, even from OSes, hypervisors, and administrators with physical access to machines. As a result, we do not suffer from the prohibitive performance and space overheads of software-based secure approaches, such as homomorphic encryption or dedicated algorithms like ASPE.

As part of our experiments, we tested the system with different workloads and analysed the influence of cache misses and page faults on the code running within secure enclaves. While both events introduce some overhead (as compared to insecure matching outside the enclave), performance degrades much more heavily with the latter, which occurs when exceeding the available amount of protected memory. We also compared the performance of SGX against the software-based ASPE alternative and observed that SGX performs systematically better as long as memory usage is kept below $93.5\,\mathrm{MiB}$.

Next, we proposed a lightweight framework for implementing secure MapReduce applications in untrusted environments (Section 3.2.2). From the user's perspective, our approach does not require any particular programming knowledge of cryptographic mechanisms or communication aspects of data distribution. Also, preserving privacy and integrity is not dependent in any way on the specific characteristics of *map* and *reduce* functions, which are defined and easily maintainable by relying on a standard Lua interpreter for code execution (Section 3.2.1). In the security aspect, we simply took advantage of isolation guarantees provided by SGX enclaves.

Our objective was to show the viability of such batch processing framework and to assess its behaviour under different conditions. We observed a correlation between the number of cache miss occurrences and the slow down of SGX executions in comparison to native ones. Besides, based on our results, we established an upper bound limit of memory usage after which it would be advisable to horizontally scale out in order to minimise the referred slow down.

We moved on to secure stream processing by introducing the design and evaluation of SecureStreams, a concise middleware framework to implement, deploy and evaluate processing pipelines for continuous data streams (Section 3.2.3). We reused our Lua port that operates in SGX enclaves with a couple adaptations aiming at interacting with a vanilla interpreter that operates outside enclaves. This, in turn, runs libraries for communication message queuing and reactive programming.

Empirical results based on real-world traces showed performance penalties of $2-3\times$ when using enclaves. We also assess SecureStreams scalability capabilities. In our setup, it reaches theoretical speed-ups as long as the number of worker nodes is kept below the total amount of processing units.

All in all, we clearly noticed performance deterioration when exhausting different memory levels: first, the L3 cache and later, the EPC limit. On the bright side, this limitation can be overcome through horizontal scalability or future hardware evolutions of SGX. Security-wise, these systems' safety relies entirely on SGX and it could be compromised by trapdoors, design flaws or hardware bugs (see Section 2.2.6). Our results open the way for further research on TEE- and cloud-based distributed systems. We keep analysing different design strategies targeted at distinct scenarios in the upcoming chapters.

Chapter 4

Group communication and data sharing

Looking into the adjustments that one should make to leverage secure enclaves in distributed communication and processing systems, we observed considerable performance implications under memory-intensive scenarios. Notwithstanding, the benefits of trusted execution environments (TEEs) outreach such niche. Confidentiality and isolation can be used in less memory-eager applications.

In this direction, we turn our attention to designing or adapting cryptographic schemes in order to profit from TEEs. The principle is basically the same and in fact derived from the construction of software guard extensions (SGX) itself: it conceals a master key somewhere very hard to find (the processor die) and uses it to derive other secrets. We do the same in Section 4.1.1, where a master secret is generated within an enclave and never leaves it, thus allowing for its usage instead of more complex asymmetric encryption derivations in the context of group access control. As a consequence, we are able to lower the computational cost of an identity-based broadcast encryption (IBBE) scheme. Likewise, in Section 4.2.1, the enclave is used to harbour keys and group membership data, so that users can share information yet guaranteeing anonymity among them. Through empirical analysis, we show the practicality of both systems and discuss their outcomes.

4.1 Cryptographic group access control

Cloud storage services have largely grown in the last decade. Many approaches rely on cryptographic solutions where data is secured on the client side before reaching the storage premises [141], therefore extenuating concerns caused by the lack of trust in the cloud provider. In the case where there is only one single reader and writer, keys can be simply shared beforehand or established over public channels according to protocols like Diffie-Hellman (DH). To enable collaborative operations, however, one needs to enforce access control policies, so that only rightful users get access to keys.

A number of cryptographic constructions have been proposed for achieving access control. The simplest, known as hybrid encryption (HE), uses symmetric and public-key cryptography by employing the former on the actual data and the latter on the symmetric key [201]. The drawback is the huge amount of metadata composed by the cumulative ciphertext produced when using the public keys of each and every group member to protect the group shared key. Other approaches rely on pairing-based cryptography (PBC) as a substitute for public-key cryptography. Even though pairing-based approaches produce small metadata volume irrespective of group sizes, they suffer from important performance issues, one order of magnitude slower than public-key cryptography [123].

We present here a cryptographic access control scheme that is both computationally- and storage-efficient considering large sets of users and dynamic membership operations [16]. This is achieved by cutting the computational complexity of an IBBE construction [202], since we profit from the isolation guarantees provided by TEEs and use a *master key* for performing membership updates. When users need to perceive such changes though, we still would obtain high overheads considering that we assume they may not have access to hardware protection (*e.g.*, mobile devices, Internet of things (IoT), distinct chip manufacturers).

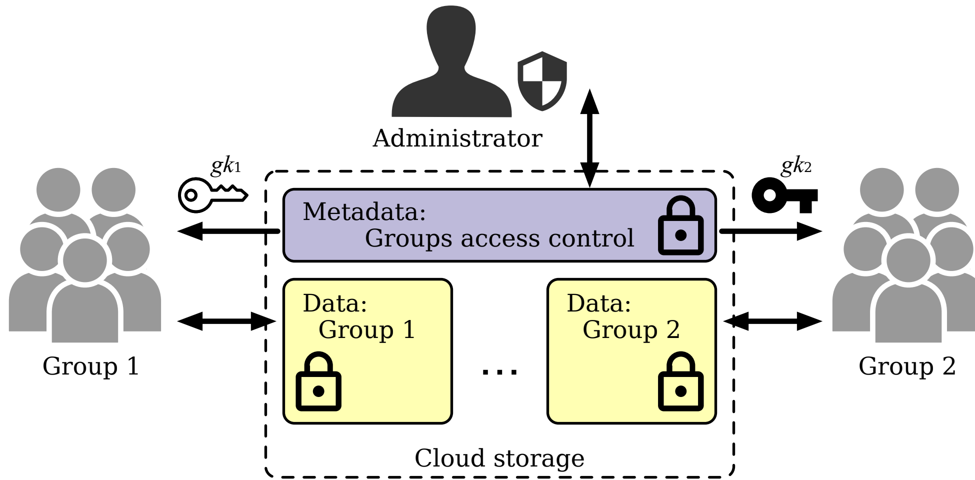

Figure 4.1: IBBE-SGX model diagram.

To mitigate this, we propose a group partitioning mechanism that imposes a complexity upper bound on the user side, limited by the partition size.

We implemented our access control scheme using Intel SGX as TEE. To do so, we adapted a PBC library [203] and its underlying dependency, GNU multiple precision arithmetic library (GMP) [204], to run within SGX enclaves. Our evaluation shows that our scheme performs better than HE with regard to metadata expansion, group creation and user removal from a group; and worse for user addition to a group and key decryption time. The benefits of having small metadata, though, goes beyond access control latencies, since it leads to less storage and network usage. Additionally, the fact that we run the access control inside secure enclaves prevents curious administrators to snoop on group keys, an extra security feature not present in traditional HE.

Even though the main motivation for this work is to securely share data in cloud environments, the proposed solution can be applied for encrypting arbitrary information that is securely broadcast to a group of users in any shared media, *e.g.*, peer-to-peer networks and pay-per-view TV. In the remaining of this section, we (i) propose a new approach to IBBE encryption by relying on Intel SGX; (ii) propose an original partitioning scheme that lowers the time required by users to absorb access control changes; (iii) implement and evaluate our system in a realistic setup; and (iv) compare it with state-of-the-art solutions.

Model

IBBE-SGX targets at managing groups of users who perform collaborative editing on cryptographically protected data stored on untrusted cloud storages. Data are protected using a block cipher encryption algorithm such as advanced encryption standard (AES), using a symmetric group key gk that is derivable by every legit group member based on metadata accessible to him.

As illustrated in Figure 4.1, we distinguish three actors:

- *administrators*, who perform membership operations: group creation, group members' addition and revocation. They may express an honest-but-curious behaviour by correctly performing their duties and maliciously trying to spy on group keys;
- *clients*, who derive keys for groups to which they belong and use them to add or modify data in the group's shared content. They are trustworthy, meaning that they do not disclose group keys; and
- *cloud storage*, which stores metadata that hold group access control information along with encrypted shared files. Besides the role of storage medium, we also use the cloud storage as communication channel between administrators and users. When the formers update membership information, this action is propagated through the respective changes in the storage. Users, in turn, may listen to modifications in files that are relevant to them. The cloud storage may try to eavesdrop on users' data and collude with administrators or revoked users.

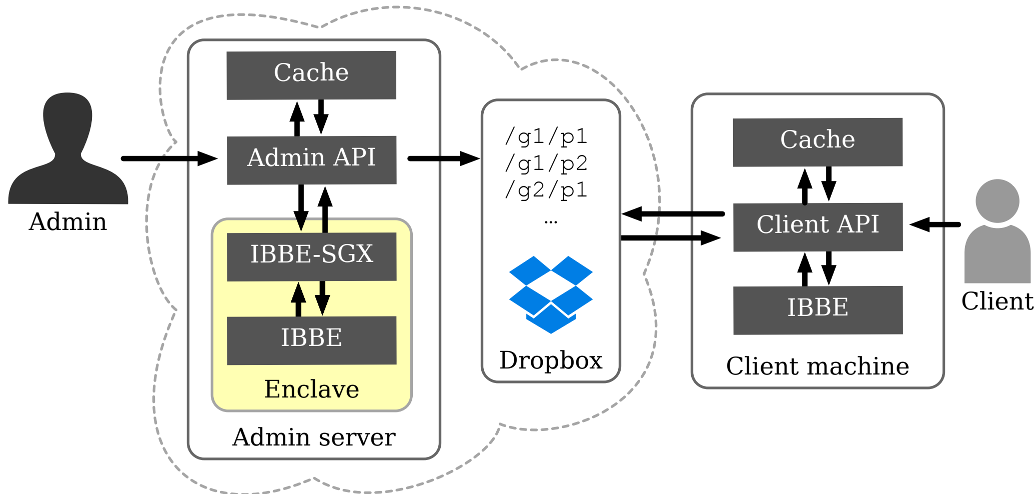

Figure 4.2: IBBE-SGX: System components.

We enforce authenticity only with respect to administrator identities on membership operations. Managing encryption and authenticating data created or altered by users is out of scope. Identities of group members are not secret, nor the type of membership operations, as they can be inferred by the cloud storage from traffic access patterns. Privacy constructions offering such guarantees [205, 206, 207] are orthogonal to our work. We propose a system that considers members anonymity in Section 4.2.1.

Figure 4.2 illustrates the system components, including a client and an administrator which use Dropbox as public cloud storage provider. The administrator's application programming interface (API) makes calls to the underlying SGX enclave that contains IBBE-SGX. This, in turn, uses the IBBE component. Clients are not required to have a TEE, although that could enhance the protection of their private key. IBBE decrypting functionalities are directly called by the client's API. Both administrators and clients may use local caches in order to save round-trips to the storage provider. Administrators use the hypertext transfer protocol (HTTP) verb `PUT` to send data to the cloud, whereas clients listen to updates on relevant files through HTTP *long polling*. In Dropbox, long polling works at the directory level. As a consequence, we index group metadata in a two-level hierarchy: parent folders represent one group, and each child a partition.

4.1.1 IBBE-SGX

IBBE-SGX access control scheme can be broadly described in 3 steps: (i) trust establishment and private key provisioning; (ii) membership definitions and group key provisioning; and (iii) membership changes and key updates. IBBE schemes generate a single public key that can be paired with several private keys, one per user. Users, in turn, need to be sure that the private key they receive is indeed generated by someone they trust, otherwise they would be vulnerable to malicious entities trying to impersonate the key issuer. To achieve that, we rely upon a public key infrastructure (PKI) to provide verifiable private keys to users.

Another security requirement of IBBE-SGX is that the key management must be kept in a TEE, so that the master key is never accessible by attackers (not even administrators). Therefore, there must be a way of checking whether that is the case. On that front, Intel SGX makes it possible to attest enclaves (Section 2.2.3). Running this procedure gives the assurance that a given piece of binary code is truly the one running within an enclave, on a genuine Intel SGX processor.

Figure 4.3 illustrates the initial setup of trust that must be executed at least once before any key leaves the enclave. Initially, the enclave generates a pair of asymmetric keys. While the private one never leaves the trusted domain, the public key is sent along with the enclave measurement to the Auditor ❶, who is both responsible for attesting the enclave and signing its certificate, thus also acting as a certification authority (CA). Next, the Auditor checks with Intel attestation service (IAS) ❷ if the enclave is genuine. Being the case, it compares the enclave measurement with the expected one, so that it can be sure that the code inside the shielded execution environment is trustworthy. Once that is achieved, the CA issues the

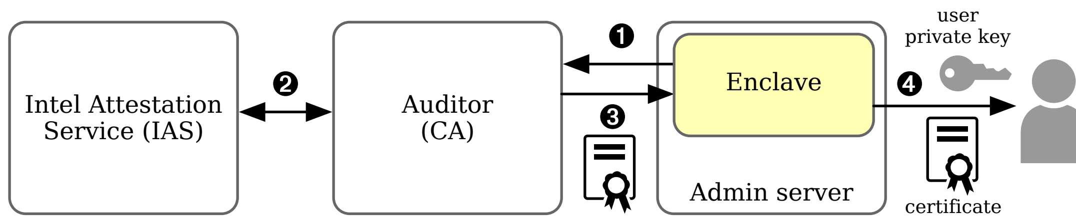

Figure 4.3: IBBE-SGX initial setup.

enclave's certificate ❸, which also contains its public key. Finally, users are able to receive their private keys and the enclave's certificate ❹. Users' keys will be encrypted by the enclave's private key generated in the beginning. To be sure they are not communicating with rogue key issuers, users check the CA's signature in the certificate and then use the enclave's public key contained therein. All communication channels described in this scheme must be encrypted by cryptographic protocols such as transport layer security (TLS).

Once user private keys are established, they can be used to retrieve group keys from metadata held in a shared storage. This can only happen, though, if the administrator has first included a given user as a member of some group and updated the respective group metadata in the storage. Likewise, if metadata were updated after a user revocation, such user will not be able to derive the group key any longer.

Cryptographic operations

Suppose that we want to come up with a simple, yet secure, cryptographic scheme to protect a group key gk by using an asymmetric encryption primitive [208], based on Rivest–Shamir–Adleman (RSA) or elliptic curve cryptography (ECC). As each user in the system has a public-private key pair, the scheme consists in encrypting gk using the public key of each member in the group. Group members can access gk by decrypting the resulting ciphertext using their private key. This construction is referred to as trivial broadcast encryption [134], or HE [123].

In such scheme, the amount of group metadata grows linearly with the number of members in the group, making it impractical in the context of very large groups. Besides, when revoking group members, a new key gk needs to be created. As a consequence, the entire group metadata needs to be regenerated and updated. This causes the propagation of the linear increase both to the cipher generation and to data transmission latencies.

Additionally, when performing group membership operations, administrators must check the authenticity of public keys that are linked to members' identity. A PKI [208] can be used to solve this issue. Apart from risks that PKI brings [209], one needs to account for the practical costs of setting up, running and accessing a PKI.

Alternatively, one could replace public-key primitives with identity-based ones. Using identity-based encryption (IBE) [210, 211] allows for the adoption of arbitrary strings (*e.g.*, user name or e-mail) as public keys, alongside public known parameters. It is even possible to encrypt messages which are addressed to users who have not yet interacted with the system. The user secret key is generated at setup phase or later by a trusted authority (TA), which guards a private master key for that purpose.

In conclusion, HE combines symmetric (AES) and asymmetric (RSA or ECC) encryptions for group communication. It shows a linear growth on metadata with respect to the number of group members. RSA requires PKI for checking authenticity of each user's public key, whereas IBE replaces the PKI for publicly known parameters plus a user identifier string, therefore obviating the need of checking the public key authenticity for every user.

Identity-based broadcast encryption (IBBE)

Broadcast encryption (BE) [137] allows a node to encrypt a message that is only accessible to a subset of all users who were provided with a private key when they initially joined the system. In such scenario, a system-wide public key (as opposed to one per user as in HE) is used to encrypt messages, which typically

contain a group key gk. The cipher is then decrypted by private keys and used by rightful group members in order to get access to protected content.

There are BE solutions that tolerate any coalition of illegitimate members [143] or allow dynamic changes of broadcast groups [212]. We chose, however, IBBE [202] that besides contemplating both features, also integrates with IBE, therefore taking advantage of small sized metadata: the system-wide public key is linear to the maximum group size, while ciphertexts and private keys have constant sizes. The drawback, however, is the quadratic complexity of crypto operations in the number of group members. In short, even though the scheme brings a tremendous gain in the size of group metadata, the computational cost of IBBE might render it unpractical.

Figure 4.4 compares hybrid encryption with public key (HE-PKI) (RSA-2048, not considering signature check), hybrid encryption with identity-based encryption (HE-IBE) (as in [210]) and IBBE schemes when encapsulating a $32\,\mathrm{B}$ key. We see, on the left, the total time taken for group creation when varying the number of members and, on the right, the group metadata expansion. IBBE always produces $256\,\mathrm{B}$ of metadata, regardless of the group members amount. That is preferable when compared to HE-PKI and HE-IBE, which produce increasingly larger values: $27\,\mathrm{MiB}$ for groups of 100,000 users, and $274\,\mathrm{MiB}$ for the largest group size. On the other hand, IBBE performs much worse than HE-PKI when considering the execution time. It is $150\times$ and $144\times$ slower for groups of 10,000 and 100,000 users, respectively.

It is clear that IBBE would be a better choice if it were not so slow. Since we can take advantage of the shielding provided by TEEs, we are able to improve its performance during encryption. Moreover, we propose a mitigation technique for lower decryption times, when we assume there is no TEE available. We succinctly explain how traditional IBBE works before introducing the proposed modifications. Let

- $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ be a bilinear map defined by cyclic groups of prime order p.
- $g \in \mathbb{G}_1$, $h \in \mathbb{G}_2$ and $\gamma, k \in \mathbb{Z}_p^*$ be random values. Besides, $w = g^\gamma$ and $v = e(g, h)$.
- n be the largest possible group size.
- $\mathcal{H} : \mathbb{Z}^* \rightarrow \mathbb{Z}_p^*$ be a cryptographic hash function

The IBBE scheme [213, 202] consists of the following operations.

1. **System setup**: TA runs it once by generating a master secret key M_K and a system-wide public key P_K.

$$M_K = \{g, \gamma\} \tag{4.1}$$

$$P_K = \{w, v, h, h^\gamma, h^{\gamma^2}, ..., h^{\gamma^n}\} \tag{4.2}$$

2. **Extract user secret**: TA uses the M_K to extract the secret key U_K for each user.

$$U_K = g^{(\gamma + \mathcal{H}(u))^{-1}}$$

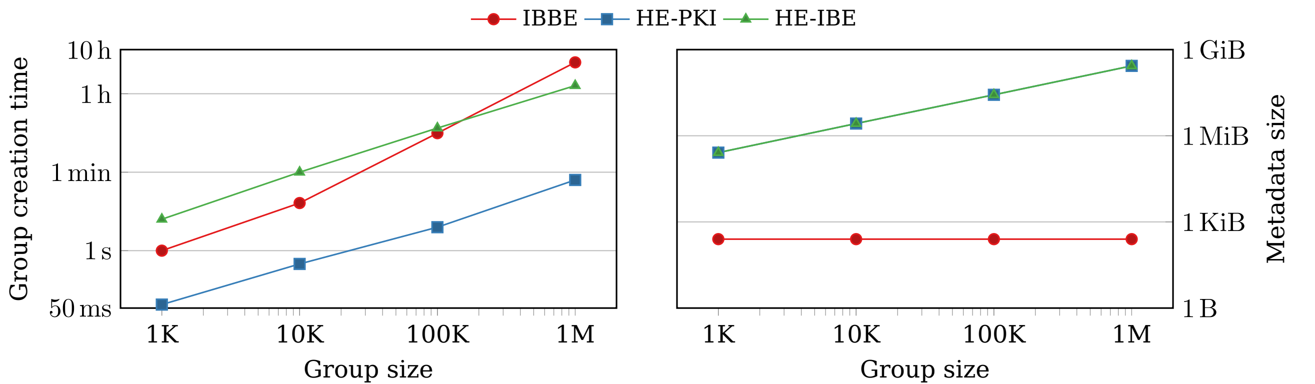

Figure 4.4: Comparison between HE-PKI, HE-IBE and IBBE regarding group creation time and metadata produced.

3. **Encrypt broadcast key**: The broadcaster generates a randomized broadcast key bk for a given set of receivers $\mathcal{S}$. Along with P_K, this operation turns bk into a public broadcast ciphertext $\mathcal{C}$.

$$
\begin{aligned}
bk &= v^k \\
\mathcal{C} &= \{C_1, C_2\}, \; where: \\
C_1 &= w^{-k} \\
C_2 &= \left(\left(h^{\gamma^N}\right) \cdot \left(h^{\gamma^{N-1}}\right)^{\mathcal{E}_1} \cdot \left(h^{\gamma^{N-2}}\right)^{\mathcal{E}_2} \cdot ... \cdot \left(h^{\gamma}\right)^{\mathcal{E}_{N-1}}\right)^k, \; where: \\
N &= |\mathcal{S}| \\
\mathcal{E}_1 &= \sum_{u \in \mathcal{S}} \mathcal{H}(u) \\
\mathcal{E}_2 &= \sum_{u_1, u_2 \in \mathcal{S}, u1 \neq u2} \mathcal{H}(u_1) \cdot \mathcal{H}(u_2) \\
&... \\
\mathcal{E}_{N-1} &= \prod_{u \in \mathcal{S}} \mathcal{H}(u)
\end{aligned}
$$

4. **Decrypt broadcast key**: any member of $\mathcal{S}$ can derive bk from $(\mathcal{S}, \mathcal{C})$ using U_K.

In contrast to traditional IBBE that requires a TA to perform the *system setup* and *extract user secret* operations, we use SGX enclaves instead. By doing so, the master secret key M_K can be used in plaintext form inside the enclave, and securely sealed when stored outside for persistence.

The operation to *encrypt* the broadcast key rely on the system-wide public key P_K, hence it can be performed by any user of the system in traditional IBBE. We, instead, require that all group membership changes and group key encryption are performed by an *administrator*. Since administrators operate on the same SGX machines where system setup and user key generation take place, encryption in IBBE-SGX may use M_K instead of P_K, which dramatically reduces the computational complexity of such operation. The decryption operation, however, remains identical to the traditional IBBE approach, executable by any user.

By using M_K inside the enclave, we are able to bypass the polynomial expansion of quadratic cost shown in step 3 of IBBE. As a consequence, the encryption operation's complexity drops from $O(N^2)$ in IBBE to $O(N)$ in IBBE-SGX. The value C_2 is simply computed by:

$$C_2 = h^{k \cdot \prod_{u \in \mathcal{S}} (\gamma + \mathcal{H}(u))} \tag{4.3}$$

This complexity cut is sufficient to tackle the IBBE's impracticality highlighted in Figure 4.4. Moreover, operations for re-keying, adding or removing a user from a broadcast group are done in $O(1)$. In order to remove a user and update the broadcast key bk in constant time, we had to add a third component $C_3 = (C_2)^{k^{-1}}$ to the ciphertext, which is completely safe since it may be entirely derived from public key's components (Equation (4.2)). This is due to the fact that the administrator has no access to the random component k which is required for updating the cipher upon key update, which happens in both user removal and re-key operations. Table 4.1 shows how to recompute the ciphertext for each of them.

Unfortunately, IBBE-SGX maintains the quadratic complexity when a group member needs to decrypt the ciphertext by performing a polynomial expansion similar to the encrypt operation described above. We address this issue by introducing a partitioning mechanism.

Partitioning

As the decryption time is bound to the number of users in the receiving set, we split the group into m partitions (sub-groups) of at most n users each, thus limiting the user decryption time to the number of members in a single partition. Each partition $p \in \mathcal{P}$ corresponds to a broadcast group such that $m = |\mathcal{P}|$ with its respective broadcast key bk_i encapsulated in a ciphertext $\mathcal{C}_i$, where $i \in \{1, \ldots, m\}$. Such key bk_i is

Table 4.1: IBBE-SGX: Constant time operations.

Add user u_a	Remove user u_r and update key to k_n	Re-keying with a new k_n
$C_2 \leftarrow (C_2)^{\gamma+\mathcal{H}(u_a)}$ $C_3 \leftarrow (C_3)^{\gamma+\mathcal{H}(u_a)}$	$C_1 \leftarrow w^{-k_n}$ $C_3 \leftarrow (C_3)^{(\gamma+\mathcal{H}(u_r))^{-1}}$ $C_2 \leftarrow (C_3)^{k_n}$	$C_1 \leftarrow w^{-k_n}$ $C_2 \leftarrow (C_3)^{k_n}$

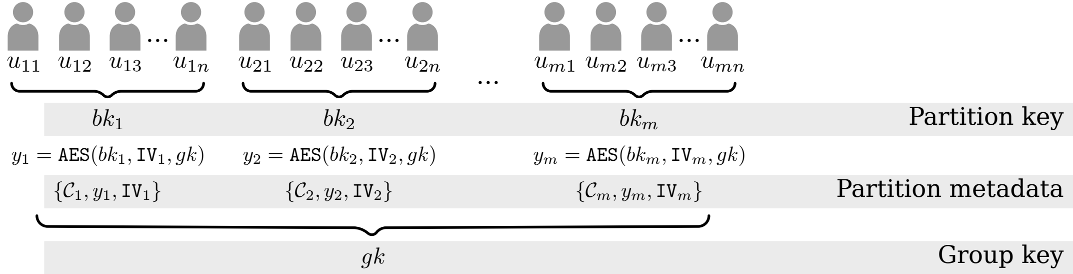

Figure 4.5: IBBE-SGX: Partitioning mechanism.

used to encrypt the group key gk shared across all partitions. To encrypt gk, we use symmetric encryption, such as AES, and produce a new component $y_i = \texttt{AES}(bk_i, \texttt{IV}_i, gk)$ where $\texttt{IV}_i$ is a random input vector, which are both appended to the partition metadata, *i.e.*, $\{\mathcal{C}_i, y_i, \texttt{IV}_i\}$. A partition p groups a set of users $u_{ij} \in p$ such that $n = |p|$ and $j \in \{1, \ldots, n\}$. Figure 4.5 illustrates the scheme.

From the administrator perspective, there is an impact when removing a user from the group, since that would provoke an update on the group key gk and therefore the recalculation of y_i for all partitions, along with bk_i of the implicated one. This renders the complexity $O(m)$ to the remove user operation instead of $O(1)$ with no partitioning. Diversely, all other operation complexities are kept or reduced. Table 4.2 compares them. In order to distinguish cardinalities, we use $n = |p|$ for partitions and $N = |\mathcal{S}|$ for a single broadcast group, *i.e.*, without the partitioning scheme.

Extracting a user key and adding a user to a group remain $O(1)$. The addition of a user can cause the creation of a new partition if all existing ones are full. In such case, the constant complexity is kept since it corresponds to that of encrypting the broadcast key bk_i for the new partition with a single member (Equation (4.3)). Creating a group has the cost of creating the first partition, or $O(n)$ (Equation (4.2)). The biggest gain comes, however, in key decryption. Instead of being quadratic in the total number of users $O(N^2)$, it becomes quadratic in the number of users per partition $O(n^2)$.

Partitioning also impacts storage footprint. The public key P_K is linear in the maximal number of users per partition $O(n)$ instead of the total user amount $O(N)$. Concerning group metadata, the footprint is augmented by the symmetrically encrypted group key, *i.e.*, y_i, and the respective initialisation vector (IV). For an entire group, metadata storage corresponds to the number of partitions times the size of $\{\mathcal{C}_i, y_i, \texttt{IV}_i\}$,

Table 4.2: IBBE-SGX: Operations complexities comparison.

Operation	IBBE [202]	IBBE-SGX	IBBE-SGX with partitioning
System setup	$O(N)$	$O(N)$	$O(n)$
Extract user key	$O(1)$	$O(1)$	$O(1)$
Create group key	$O(N^2)$	$O(N)$	$O(mn)$
Add user to group		$O(1)$	$O(1)$
Remove user from group		$O(1)$	$O(m)$
Decrypt group key	$O(N^2)$	$O(N^2)$	$O(n^2)$

Cardinalities: N: global number of users. n: members in one partition. m: number of partitions.

Algorithm 1: IBBE-SGX: Create group.

input : g: group
$\mathcal{S} = \{u_1, ..., u_N\}$: members
n: partition size

1 $\mathcal{C} \leftarrow \emptyset$
2 $\mathcal{P} \leftarrow \{\{u_1, ..., u_n\}, \{u_{n+1}, ..., u_{2n}\}, ...\}$

Enclaved

3 $gk \leftarrow RandomKey()$
4 **foreach** $i \in \mathcal{P}$ **do**
5 $\quad \{bk_i, \mathcal{C}_i\} \leftarrow create_partition(M_k, i)$
6 $\quad \mathtt{IV}_i \leftarrow RandomIV()$
7 $\quad y_i \leftarrow \mathtt{AES}(\mathtt{SHA}(bk_i), \mathtt{IV}_i, gk)$
8 $\quad \mathcal{C} \leftarrow \mathcal{C} \cup \{i, \mathcal{C}_i, y_i, \mathtt{IV}_i\}$
9 $sealed_gk \leftarrow sgx_seal(gk)$

10 $store_key(g, sealed_gk)$
11 $\forall c \in \mathcal{C} : store_meta(g, c)$

Algorithm 2: IBBE-SGX: Add user to group.

input : g: group
$\mathcal{P}$: partitions of g
n: partition size
u_a: user to add
$sealed_gk$: sealed group key

1 $p_a \leftarrow \exists p \in \mathcal{P},\ such\ that\ |p| < n$
2 **if** $p_a = \emptyset$ **then**
3 $\quad p_a \leftarrow \{u_a\}$

Enclaved

4 $\quad \{bk_a, \mathcal{C}_a\} \leftarrow create_partition(M_k, p_a)$
5 $\quad gk \leftarrow sgx_unseal(sealed_gk)$
6 $\quad \mathtt{IV}_a \leftarrow RandomIV()$
7 $\quad y_a \leftarrow \mathtt{AES}(\mathtt{SHA}(bk_a), \mathtt{IV}_a, gk)$

8 $\quad store(g, \{p_a, \mathcal{C}_a, y_a, \mathtt{IV}_a\})$
9 **else**
10 $\quad p_a \leftarrow p_a \cup \{u_a\}$
11 $\quad \mathcal{C}_a \leftarrow add_user_to_partition(M_k, p_a, u_a)$
12 $\quad update_meta(g, \{p_a, \mathcal{C}_a, -, -\})$

in addition to a data structure that keeps the mapping between users and partitions. Although this induces a slight overhead, the number of partitions in a group is relatively small compared to the group size. Besides, administrators alone manipulate partition metadata. They can therefore use a local cache to bypass the cost of accessing remote storage.

Determining the optimal value for the partition size mostly depends on the dynamics of the group. There is a trade-off between the number and frequency of group membership operations performed by the administrator and those performed by regular users for decrypting the broadcast key. A small partition size reduces the decryption time on the user side while a larger one reduces the number of operations performed by the administrator to run IBBE-SGX and to maintain the metadata. We further evaluate this trade-off in the upcoming sections.

Membership operations

In order to create a group, we execute the instructions in Algorithm 1. Once the partitions are determined (line 2), the execution enters the SGX enclave (lines 3 to 9), when the group key is encrypted with the partition broadcast key. The ciphertext and the sealed group key leave the enclave to be later pushed to the shared storage (lines 10 and 11).

The operation of adding a user to a group, shown in Algorithm 2, starts by finding one partition which is not yet full (line 1). If none is found, a new partition is created with a single user (line 3) and the group key is enveloped by the broadcast key of this new partition (lines 4 to 7), before storing the corresponding ciphertexts (line 8). Otherwise, the user is added to the not-yet-full partition found (lines 10 and 11). Since both the partition broadcast and the group key remain unchanged, only the ciphertext needs to be updated (line 12).

When a user needs to be removed from a group (Algorithm 3), his identifier is extracted from the partition to which he belongs (lines 1 and 2) and a new group key is randomly generated (line 3). Next, metadata from the partition that used to contain the removed user is updated (line 4) and the new partition key bk_r is used for enveloping the new group key gk (line 6). For all the remaining partitions, a constant time re-keying operation (see Table 4.1) regenerates each partition broadcast key, which is used to encrypt the new group key (lines 8 to 10). After sealing the new group key (line 11), the resulting value is persisted (line 12) along with all updated metadata (lines 13 and 14).

Algorithm 3: IBBE-SGX: Remove user from group.

input : g: group
$\mathcal{P}$: partitions of g
u_r: user to remove

1 $p_r \leftarrow p \in \mathcal{P},\ \textit{such that}\ u_r \in p$
2 $p_r \leftarrow p_r \setminus \{u_r\}$

Enclaved

3 $gk \leftarrow RandomKey()$
4 $(bk_r, \mathcal{C}_r) \leftarrow remove_user(M_k, p_r, u_r)$
5 $\texttt{IV}_r \leftarrow RandomIV()$
6 $y_r \leftarrow \texttt{AES}(\texttt{SHA}(bk_r), \texttt{IV}_r, gk)$
7 **for** $p \in \mathcal{P} \setminus p_r$ **do**
8 $\quad (bk_p, \mathcal{C}_p) \leftarrow rekey_partition(p)$
9 $\quad \texttt{IV}_p \leftarrow RandomIV()$
10 $\quad y_p \leftarrow \texttt{AES}(\texttt{SHA}(bk_p), \texttt{IV}_p, gk)$
11 $sealed_gk \leftarrow sgx_seal(gk)$

12 $update_key(g, sealed_gk)$
13 $update_meta(g, \{p_r, \mathcal{C}_r, y_r, \texttt{IV}_r\})$
14 $\forall p \in \mathcal{P} \setminus p_r : update_meta(g, \{p, \mathcal{C}_p, y_p, \texttt{IV}_p\})$

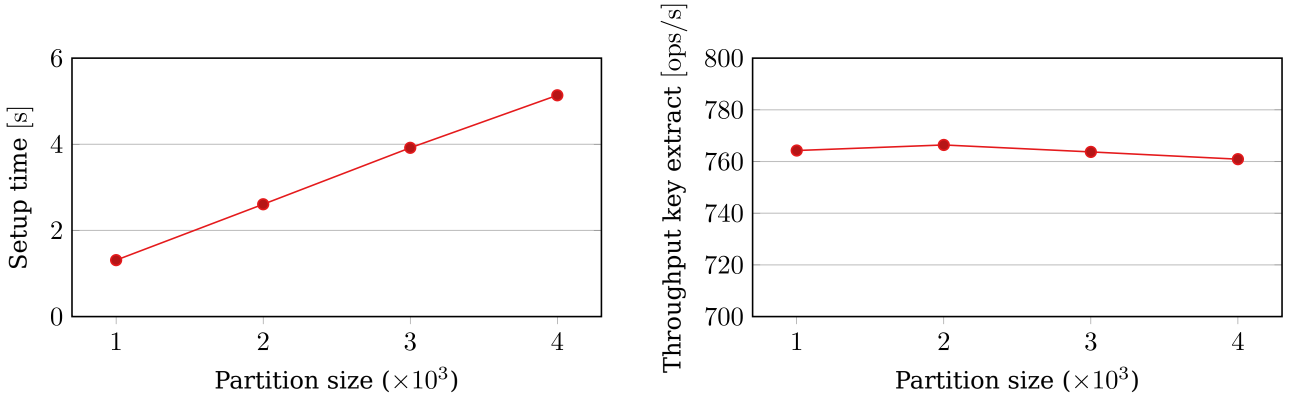

Figure 4.6: Performance of IBBE-SGX's bootstrap phase.

As many removal operations may cause sparsely occupied partitions, we propose a re-partitioning scheme whenever partition occupancies are low. We implement a heuristic to detect low occupancy when half of the partitions in one group are two thirds full or less. In such case, re-partitioning is triggered by simply re-creating the implicated group according to Algorithm 1. The client decrypt operation works by first using the standard IBBE to discover the broadcast key, whose hash is then used to obtain the group key.

Implementation and evaluation

We used the PBC [203] library which depends on GMP [204]. They both have to be used inside SGX enclaves to implement the IBBE component (see Figure 4.2). Luckily, since both PBC and GMP mostly perform computations rather than input and output operations, the challenges on adapting them were chiefly restrained to tracking and adapting calls to GNU C library (glibc). The adaptations were done either by relaying operations to the operating system through outside calls (ocalls), or performing them with equivalent operations inside enclaves. Outside calls, in turn, do not perform any sensitive action that could compromise security. Aside from source code, we also adapted the compilation toolchain, since enclaves must use curated versions of standard libraries (we used Intel SGX software development kit (SDK)). Moreover, specific code generation flags must be used, along with the prevention of using the compiler's built-in functions. The modifications account for 32 line of code (LoC) for PBC and 299 for GMP.

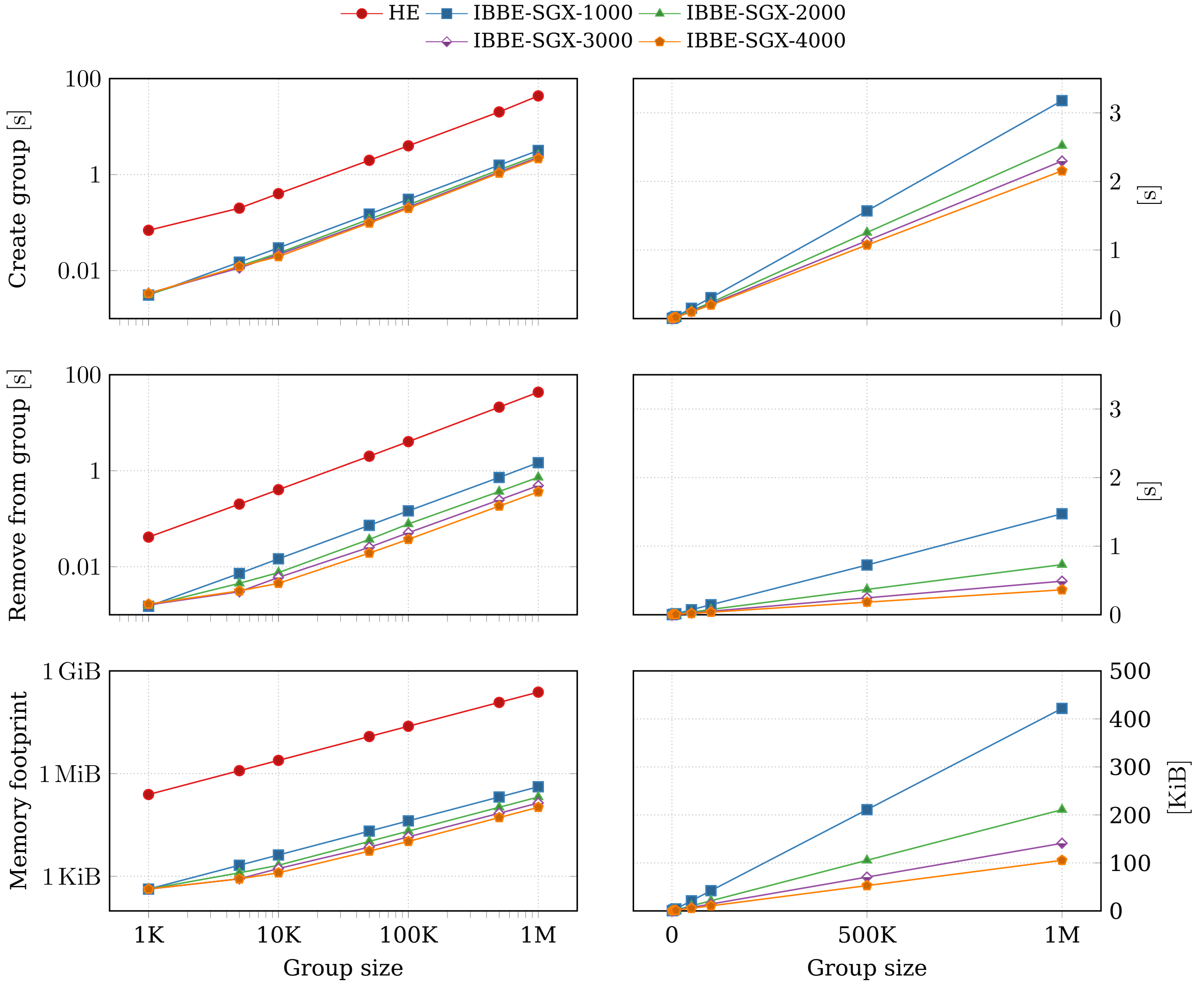

Figure 4.7: Evaluation of create and remove operations and storage footprint, by varying the partition size for IBBE-SGX (1000, 2000, 3000 and 4000).

Apart from these changes, we also needed to use common cryptographic libraries. Due to limitations in Intel SGX SDK v.1.9, we used an OpenSSL SGX port [214]. The end-to-end system including both IBBE-SGX and HE schemes consists of 3,152 lines of C/C++ code and 170 lines of Python.

Micro-benchmarks

We benchmark the performance of IBBE-SGX in isolation, by comparing it to HE, and by using access control traces. Experiments were performed on a quad-core Intel i7-6600U machine with a processor at 3.4 GHz, 16 GB of random-access memory (RAM) and Ubuntu 16.04 long term support (LTS).

First, we evaluate the bootstrap's performance. It consists of setting up the system and generating user private keys. Results are shown in Figure 4.6. The setup latency increases linearly according to partition size, with a growth of $1.2\,\mathrm{s}$ per 1.000 users. Differently, extracting secret user keys gives an average throughput of $764\,\mathrm{ops/s}$, irrespective of the partition size. This is unsurprising, since we have seen that such operations are $O(n)$ and $O(1)$, respectively (see Table 4.2).

Next, we compare IBBE-SGX with HE. Figure 4.7 displays performance measurements of operations for creating a group and removing a user from a group, along with the storage footprint of metadata. IBBE-SGX is better than HE in all three by approximately a constant factor. Create and remove operations with

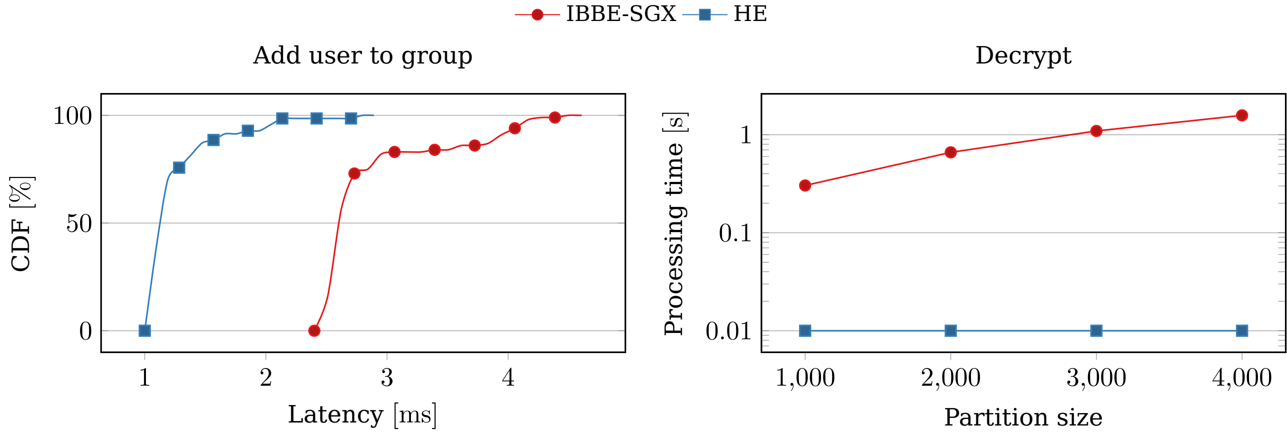

Figure 4.8: Performance of adding a user to a group and decrypt operations.

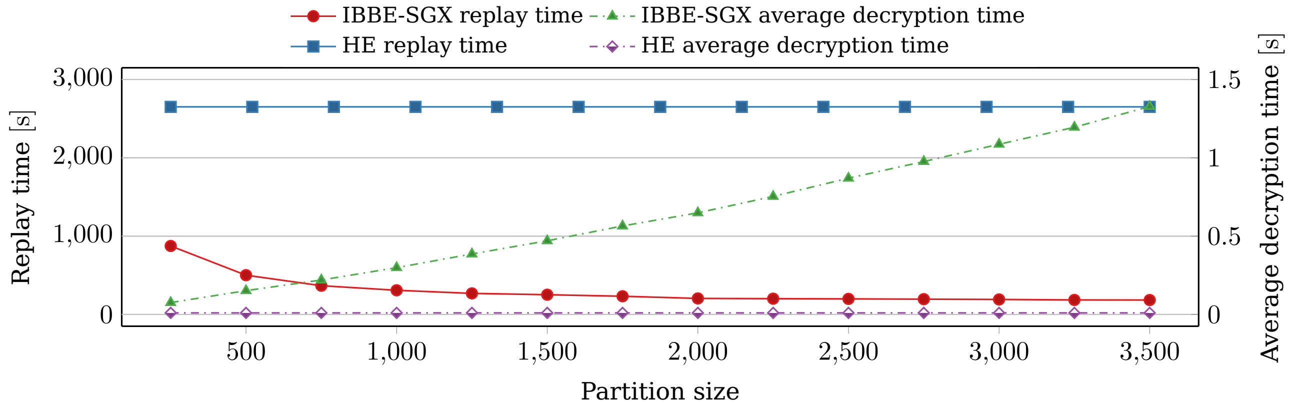

Figure 4.9: Measuring total administrator replay time and average user decryption time per different partition sizes using the Linux Kernel access control list data set.

IBBE-SGX is 1 order of magnitude faster than HE. If we compare to the original IBBE scheme, IBBE-SGX is better by 2 orders of magnitude for groups of 1,000 users and 3 for one million users (see Figure 4.4). Concerning memory usage, IBBE-SGX is up to 6 orders of magnitude better than HE. The plots on the right-hand side in Figure 4.7 show the same data in linear axis but disregarding HE, so that we are able to observe the effect of different partition sizes. One can notice that the remove operation takes half the time of creating a group. Considering the storage footprint, the increase in memory use brought by using smaller partition sizes is fairly small. For instance, 422 KiB with 1000 users per partition vs. 105 KiB with 4000 users per partition, both for groups with 1 million members.

The cumulative distribution function (CDF) of latencies for adding a user to a group is shown in Figure 4.8. The operation has a constant time complexity for both IBBE-SGX and HE. As noted in Algorithm 2, the user addition operation in IBBE-SGX can take two paths: to an existing partition or to a new one if all the others are full. Such effect can be perceived in the plot from 80th percentile on. Besides, the HE add operation is generally twice as fast as IBBE-SGX.

The client decryption performance is shown on the right-hand side in Figure 4.8. This operation is also faster with HE. The difference of 2 orders of magnitude is caused by the quadratic cost of IBBE-SGX decryption operation. We argue that a slower decryption time for IBBE-SGX can be acceptable in practice, since it is preceded by metadata updates which involve cloud communication and is therefore minimised when put into perspective. Additionally, the decryption cost is bounded to the partition size, no matter the number of users in the group.

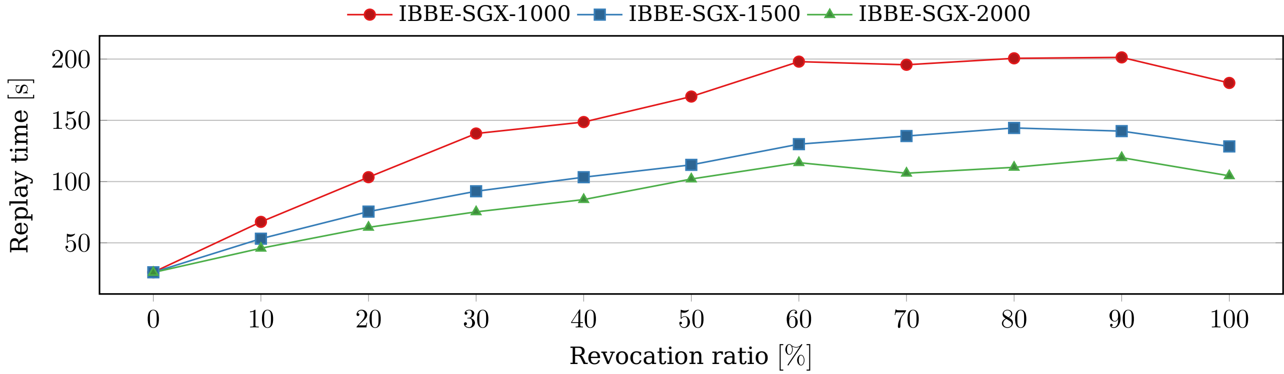

Figure 4.10: IBBE-SGX: Replay time of synthetic datasets varying partition sizes and revocation rates.

Macro-benchmarks

To capture the performance of the IBBE-SGX scheme within a realistic scenario, we replay an access control trace based on the membership changes in the version control repository of the Linux Kernel [215]. We derive the membership trace by considering one single group where the first commit of a user represents his addition. Conversely, the last commit is interpreted as his removal from the group. This crafted trace contains 43,468 membership operations that spawn across a period of 10 years, during which the group size never exceeds 2803 users. We replay the generated trace sequentially for both HE and IBBE-SGX by varying partition sizes. We measure the total time spent by the administrator to replay the whole trace and the average user decryption time. Figure 4.9 shows the results.

Considering the administrator replay time, IBBE-SGX performs better when the partition size converges to the number of users in the group because, in this case, the overhead of maintaining partitions is minimised. Using a small partition size, *e.g.*, 250, is $2.8\times$ more inefficient when compared to partitions of 1000 users. Compared to HE replay time, IBBE-SGX is 1 order of magnitude faster for partitions of 1250 users and beyond. Contrarily, decryption time for IBBE-SGX grows quadratically according to the partition size while in HE it remains constant. This highlights the IBBE-SGX's trade-off caused by different partition sizes. Satisfactory outcomes both in terms of administrator performance and user decryption times may be hence achieved by finding a partition size (such as 750 users) that balances both metrics given the application requirements (*e.g.*, decryption times of $250\,\mathrm{ms}$).

In order to observe the impact of different workloads of group membership access control, we generated a set of synthetic traces with incremental revocation rates. Namely, eleven traces consisting of 10,000 membership operations. Each trace is composed by randomly generated operations according to different revocation rates. We replay the traces and measure the total time required by the administrator to perform all membership changes and repeat the experiment for distinct partition sizes.

Results are shown in Figure 4.10. We observe a linear increase in the total time when incrementally increasing the revocation ratio roughly up to 50%, *i.e.*, in workloads dominated by constant-time addition operations. After this point, the total time stabilizes and finally decreases when the revocation ratio is more than 90%. This behaviour is caused by the merging of sparse partitions, which happens more frequently with the increase of revocations. With fewer partitions, IBBE-SGX's remove operation becomes faster, therefore decreasing the total time.

In conclusion, although IBBE-SGX loses in performance for some operations (*i.e.*, decryption and user addition) when compared to HE, its gains arguably outweigh the drawbacks. Due to TEE usage, administrators are prevented from accessing group keys, and are therefore restricted to only perform membership operations, *i.e.*, they could not possibly inspect into groups' shared content. Apart from the operations of creating groups and removing users from groups, which are more performant in IBBE-SGX, the biggest advantage comes with metadata reduction — orders of magnitude lower than HE, which saves resources both in bandwidth and storage space. Although shared content is protected by this scheme, group membership information is not. This raises privacy concerns when the very fact of belonging to a group is

sensitive information. In order to address this issue, we propose in the following section a scheme where this is not leaked.

4.2 Anonymous file sharing

Besides shared data, identity of users may also be sensitive information. For instance, virtual data rooms (VDRs) [216] are used for exchanging confidential documents during business acquisitions. Apart from having to enforce access control, stakeholders' identities must be protected, or else they could have business strategies disclosed to competitors.

Confidential systems that focus on group communication offer anonymity guarantees by group key exchange methods [147], requiring all active members to be present and take part in a multi-phase protocol (*e.g.*, Diffie-Hellman) each time a key is derived. Such an approach is indeed suitable for instant group communication, but impractical for file sharing that generally does not require online users. Theoretical anonymous file sharing extensions have been proposed [149, 151] without ever turning into functional systems.

Anonymous sharing of confidential content was practically addressed in an unsophisticated manner by GNU privacy guard (GPG). It drops public keys from the resulting ciphertext, therefore keeping no references to the identity of recipients. The drawback comes at decryption time, when recipients must perform asymmetric decryption trials until the portion of the ciphertext matching their private key is found, if any. Due to this, GPG works well for groups of few users but quickly becomes impractical for larger ones (Section 2.5, Table 2.2).

To tackle that, we propose an anonymous access control scheme that leverages SGX for a narrow scope: enforcing anonymity during the publishing operation (*i.e.*, upon *writing*). Our scheme does not require a TEE on the user side for performing the *read* operation, nor does it require that users connect to a designated proxy. To demonstrate the feasibility of our solution, we propose a scalable system design leveraging micro-services that can possibly scale depending on the workload.

By doing so, we achieve an improvement of three orders of magnitude when compared to state-of-the-art anonymous broadcast encryption (ANOBE) [149]. Besides, our end-to-end implementation, A-Sky [18], can scale to cope with a realistic amount of administrative and user operations.

In the remaining of the section, we *(i)* define a theoretical anonymous cryptographic access control extension that relies on TEEs for a minimal subset of operations (*i.e.*, *writes* but not *reads*); *(ii)* propose an end-to-end design that leverages micro-services which can scale according to the undergoing workloads; and *(iii)* implement and evaluate the system, first in isolation showing its benefits against state-of-the-art cryptographic schemes, and then by benchmarking its scaling capabilities and practical feasibility.

Model and use case

We consider users (humans or software agents) that are uniquely identified within the premises of an organisation and share files. They are clustered into uniquely identifiable *groups* by organisation-specific policies. We consider two functional categories: *access control* and *content management*. The first represents group membership operations (adding and removing users from groups) and is performed by *administrators*, while the latter comprises file creation and consuming by group members (writes and reads). A group member can perform one or both roles of *writer* or *reader* within one or multiple groups. The cloud object storage holds uniquely-identified binary objects (*e.g.*, Amazon S3).

We define four security properties for our confidential and anonymous file sharing system:

1. **Confidentiality and authenticity**: content of shared files is exclusively accessible by group members.
2. **Forward secrecy**: compromising a group secret does not compromise past sharing sessions.
3. **Recipients privacy**: no recipient except the group administrator is able to know other recipients' identities (*i.e.*, *readers*).
4. **Sender privacy**: no recipient except the group administrator is able to know the sender's identity (*i.e.*, *writer*).

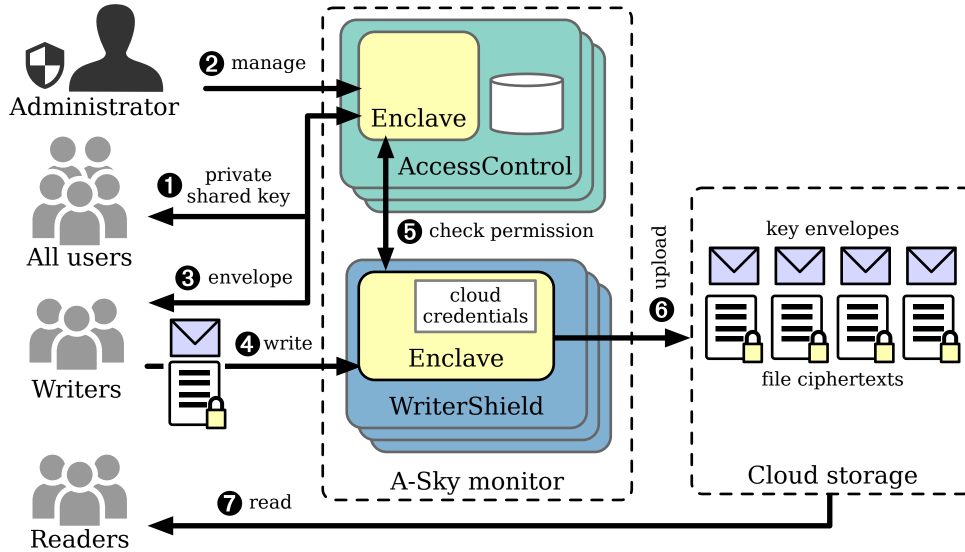

Figure 4.11: A-Sky architecture.

Revoked and external users behave arbitrarily. They may try to read shared content and identify group members by intercepting, replaying, deciphering and altering exchanged messages (*i.e.*, Dolev-Yao [217] adversarial model). User anonymity is not only threatened by external adversaries, but also internally by peer group members. Being so, active users that can rightfully decrypt group content may maliciously try to infer their peers' identities.

The storage provider is *honest-but-curious*, *i.e.*, it accordingly provides its services while possibly trying to inspect file contents and user identities (*e.g.*, by traffic analysis). Revoked users may collude with the cloud storage to get access to content shared after their revocation. Lastly, our privacy model enforces anonymity guarantees solely with respect to user identities, and not to group sizes, content lengths and traffic patterns.

Virtual data rooms (VDRs) [216], for instance, strictly control repositories of electronic documents for company mergers and acquisitions (M&A). Thanks to VDRs, sellers, supporting parties who assist them and bidders can confidentially exchange documents (*e.g.*, terms and valuations) atop of untrusted remote storage mediums. The seller acts as *administrator* and enforces access control. User roles are composed by *writers* (sellers and supporting parties) and *readers* (bidders). As enforced by confidentiality agreements, supporting parties operate in the seller's interest, and remain unidentifiable from one another. Similarly, bidders identities are concealed among themselves. As such, *inner* anonymity guarantees need to be enforced within *writers* and *readers*, while *outer* anonymity has to withstand against any external entity to the M&A process. Additionally, withdrawing bidders or misbehaving supporting parties can be *revoked* by the seller, therefore becoming unable to access the document corpus any further.

4.2.1 A-Sky

A-Sky is composed of two components: a cryptographic key management and a data delivery protocol, both designed to combine performance, data confidentiality and user anonymity. In order to avoid passing all the system operations through a SGX *monitor*, we propose a design in which only *writers* are constrained to pass through such a proxy (Figure 4.11, step ❹). *Readers* consume confidential content without communicating with the TEE-enabled monitor (❼), therefore reducing service time penalties.

The monitor acts as an outbound TA that authenticates all the content passing by. Being a proxy, it also masks the identities of data writers. Moreover, since it executes in a TEE, traditional anonymous key management schemes [149, 151] can count on a trusted entity for the key enveloping operation, therefore allowing the usage of simpler and more efficient encryption schemes.

The *monitor* sits in between end-users and the cloud storage. It is split into two micro-services, whose amount of instances can be independently adapted according to the undergoing load:

(i) the AccessControl provides a cryptographic mechanism for storing and enforcing access control to data.

Our cryptographic key management solution requires the connection to a TEE monitor. It generates keys that are securely shared with users at their first interaction with the monitor (❶). This allows the usage of symmetric encryption during the enveloping step, therefore taking advantage of hardware acceleration and smaller ciphertexts while achieving equivalent security in comparison to asymmetric approaches. The AccessControl ensure that only authorized entities can write and decrypt data, according to memberships established by an administrator (❷). Based on this, envelopes are generated with metadata that allow rightful users to decipher shared content (❸).

(ii) the WriterShield acts as an outgoing proxy for write operations.

A-Sky requires that users write the encrypted shared content through the WriterShield service (❹), which checks with the AccessControl whether a user has the permission to write in a given group (❺). Being the case, it authenticates the outgoing content and does the writing itself (❻). Since it runs in an enclave, the cloud storage credentials are safely stored by the WriterShield service.

In traditional anonymous key sharing solutions [149], a TA sets up the key management system and extracts user private keys, while end users perform key enveloping and content encryption by employing asymmetric cryptographic primitives, *i.e.*, using the public keys of group members. Differently, we leverage TEEs and are therefore able to use more efficient symmetric constructs. This change also brings advantages with respect to creating indexes that make de-enveloping more efficient, which will be described later.

Before relying on any service of the A-Sky *monitor*, it is necessary to check whether the service is running on a trustworthy Intel SGX platform and that the instances of the AccessControl and WriterShield are genuine. This validation phase is performed by administrators and users using the typical SGX attestation procedure and establishment of a secure channel (see Section 2.2.3).

System design and implementation

AccessControl

The AccessControl is responsible for generating and storing user keys, maintaining group membership information and generating envelopes. Since it deals with sensitive information, its core runs entirely within enclaves. All exchanges are encrypted with keys only known in the trusted environment: persisted state is ciphered and stored in a database while external communication is done through TLS connections terminated inside the enclave.

Its methods are invoked by regular users and administrators after secure channels are established upon successful attestation processes. All these interactions happen through a TLS-encrypted REST-like protocol. Exchanges are represented in JavaScript object notation (JSON), for which we slightly modified a C++ library [218]. In order to terminate TLS connections in the enclave, we use a port of OpenSSL for SGX [219].

Users interact with the AccessControl either to obtain *(i)* their credentials, which are randomly generated $256\,\mathrm{bit}$ secret shared keys that are stored by the monitor; or *(ii)* envelopes, to be attached to shared encrypted files. Given a unique user identifier u_{id}, the service generates a key u_k and stores it in a container $keys$ of pairs $\langle u_{id}, u_k \rangle$, *i.e.*, $keys[u_{id}] \leftarrow u_k$. Administrators, on the other hand, may create and populate groups. Depending on granted access capabilities, users can be content readers, writers or both. Such information is kept within persistent dictionaries, $group^r$ and $group^w$, which store sets of users belonging to each group identifier g_{id}, *e.g.*, $group^w[g_{id}] = \{u_a, ..., u_z\}$. Only administrators can modify these dictionaries.

Once groups are defined, metadata that grants access to files may be generated upon user requests. We call this operation *key enveloping*, and it consists of encapsulating an *access key* such that only group members with read permission are able to retrieve it. An envelope contains a file access key encrypted several times, once per group member. Just like user keys, access keys are $32\,\mathrm{B}$ long. We use AES Galois

Algorithm 4: A-Sky: AccessControl key enveloping.

```
input  : u_id: writing user identity
         g_id: group identifier
         k: file access key
output : envelope: ciphertext of the access key
1 Procedure KeyEnveloping(u_id, g_id, k):
2     envelope ← ∅
3     if u_id ∈ group^w[g_id] then
4         forall users u ∈ group^r[g_id] do
5             u_sk ← keys[u]
6             (c_k, t) ← AE(u_sk, k)
7             envelope ← envelope ∪ {(c_k, t)}
8     return envelope
```

Algorithm 5: A-Sky: Key enveloping with efficient decryption.

```
input  : u_id: writing user identity
         g_id: group identifier
         k: file access key
output : envelope: ciphertext of the access key
1  Procedure KeyEnveloping(u_id, g_id, k):
2      envelope ← ∅
3      if u_id ∈ group^w[g_id] then
4          nonce ← Random
5          forall users u ∈ group^r[g_id] do
6              u_sk ← keys[u]
7              l_u ← H(u_sk || nonce)
8              (c_k, t) ← AE(u_sk, k)
9              envelope ← envelope ∪ {(l_u, c_k, t)}
10         Sort(envelope)                    // by label l_u
11     envelope ← nonce || envelope
12     return envelope
```

counter mode (GCM), which generates a *tag* of $16\,\mathrm{B}$ for integrity. Considering the addition of $12\,\mathrm{B}$ for the IV, each group member adds $60\,\mathrm{B}$ to the envelope.

As shown in Algorithm 4, given the identity of the writing user u_{id}, the group unique identifier g_{id}, and the file access key k, the algorithm produces a ciphertext called *envelope*. The AccessControl first checks if the user u_{id} has writing permission for the group g_{id} (line 3). If it is the case, the *envelope* is built by the union of ciphertexts and authentication tags resulted from encrypting the access key k using the secret key u_{sk} of each group member u (lines 4-7).

We establish the following operations and notations for the Algorithms:

- $E(k, p) \rightarrow c$ and $D(k, c) \rightarrow p$ are symmetric encryption and decryption algorithms (*e.g.*, AES-counter mode (CTR)) using the key k, where p is plaintext and c is ciphertext;
- $AE(k, p) \rightarrow (c, t)$ and $AD(k, c, t) \rightarrow \{p, \perp\}$ are authenticated encryption and decryption algorithms (*e.g.*, AES-GCM). Besides encrypting, it also produces an authentication tag t that ensures the integrity of the ciphertext c under the key k. During decryption, in case the integrity check fails, no plaintext is generated ($\perp$).
- $S(PK^{-1}, p) \rightarrow \sigma$ and $V(PK, p, \sigma) \rightarrow \{true, false\}$ are digital signature and verification algorithms (*e.g.*, RSA-based or ECDSA) employing an asymmetric public/private key pair (PK and PK^{-1}).
- $\mathcal{H}$ is a unidirectional cryptographic hash function, *e.g.*, secure hash algorithm (SHA); and
- $||$ is the literal concatenation operation.

As in traditional ANOBE schemes [149, 151], we propose a method that reduces user decryption times by circumventing the need to perform $O(n)$ key decryption trials (line 5 of Algorithm 8) by trading it off for an increase in key enveloping times and in the envelope size (these trade-offs are evaluated later). To this end, publicly known labels are appended to each user fragment in the envelope and used as its index. Such fragments are lexicographically sorted by labels and can be hence located in logarithmic time with respect to the group size (*e.g.*, binary search). Consequently, users retrieve their keys performing one single decryption of the located fragment.

Labels are computed as the SHA-224 hash of the user's secret shared key (also known by the AccessControl service) salted with a *nonce*. This adds $28\,\mathrm{B}$ to the envelope for each group member. Our approach is more efficient in comparison to traditional ones that perform modular exponentiations [149] or tag-based encryption [151]. Algorithm 5 details the efficient variant of key enveloping with the creation of labels as

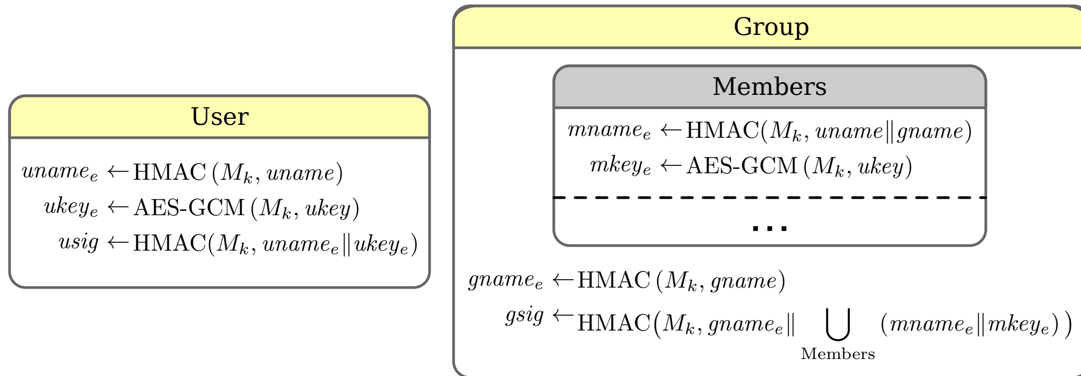

Figure 4.12: Data model of user and group *documents* stored in MongoDB.

the salted hash of the user secret key (line 7) and the fragments sorting (line 10). The random *nonce* to be used as salt is generated for each envelope (line 4) and publicly attached to it (line 12).

For implementation purposes, we split the AccessControl service into two components. The first, developed in C++ for a total of 3000 LoC, constitutes the entry-point for service requests. The other holds users and groups metadata within a replicated database. For this purpose, we use a cluster of MongoDB [220] servers with three replicas, since it offers out-of-the-box scale-out support and is well suited to store denormalized documents. In order to perform queries against it from the first sub-component, we ported the official MongoDB client library [221] to run inside an enclave. To secure the data stored in the MongoDB back-end, each replica of the entry-point sub-component is provisioned with a master key M_k at attestation time.

Since the storage back-end runs outside of enclaves, we make sure that every piece of stored data is either hashed using the keyed-hash message authentication code HMAC-SHA256 or encrypted using AES-GCM. We thus guarantee that providers that host MongoDB instances cannot infer any information about users or groups (apart from the size of each group, which is already leaked in the *envelopes*). Figure 4.12 shows how we organize data in MongoDB: one collection of users and other of groups. Users are stored once in the users collection and again in each group of which they are member. This de-normalisation prevents the service provider from inferring to which groups a user belongs, since the attributes of a given user are hashed or encrypted differently for each representation (*i.e.*, we use the name of the group as salt when hashing, and different IVs when encrypting). To ensure integrity, each document is signed using keyed-hash message authentication code (HMAC).

WriterShield

The WriterShield serves the purposes of *(i)* protecting cloud storage credentials; *(ii)* signing the ciphertext content; *(iii)* hiding user identities by proxying their requests to the cloud storage; and *(iv)* generating access tokens to the cloud storage to improve performance. When forwarding user requests to write files, the WriterShield checks with the AccessControl whether the query comes from a user who has the correct permissions.

User requests, including file contents, cross over the enclave boundary. This obviously slows down transmission rates because of content re-encryptions and trusted/untrusted edge transitions. To tackle that, we have also implemented a variant where temporary access tokens are given to users. Such tokens allow them to upload content without crossing through the TEE. The ciphertext digest, however, still needs to be authenticated by the signing key available in the WriterShield. In such case, users are responsible for using appropriate proxies that can conceal the request's origin. Also, communication with the cloud storage must happen through encrypted connections. Even if the data files are encrypted, metadata can leak group information along the network path.

We modelled the cloud storage component using Minio [222], a distributed object store that is fully compatible with the APIs of Amazon simple storage service (S3). As we need to perform operations in the cloud storage from enclaves, we ported the Java version of the Minio client library to C++ so that it can

Algorithm 6: A-Sky: WriterShield proxy file.

```
input : f: file reference
        u_id: writing user identity
        g_id: group identifier
        C: file ciphertext
        A: AccessControl instance
1 Procedure ProxyFile(f, u_id, g_id, C, A):
2     if u_id ∈ A.group^w[g_id] then
3         σ ← S(PK_TA^-1, C)
4         UploadToCloud(f, C, σ)
```

Algorithm 7: A-Sky: User write file to group.

```
input : f: file reference
        u_id: writing user identity
        g_id: group identifier
        P: file plaintext
        A: AccessControl instance
        W: WriterShield instance
1 Procedure WriteToGroup(f, u_id, g_id, P, A, W):
2     fk ← Random                          // file access key
3     envelope ← A.KeyEnveloping(u_id, g_id, fk)
                                           // i.e., Alg. 4 or Alg. 5
4     cipher ← E(fk, P)
5     C ← envelope || cipher
6     W.ProxyFile(f, u_id, g_id, C, A)
                                           // i.e., Alg. 6
```

run together with the WriterShield. These modifications account for 4000 LoC of C++. Without accounting for external libraries, the WriterShield consists of 800 LoC.

As the WriterShield is the sole service possessing the write credentials for the cloud provider, it constitutes a necessary hop for uploading the file: either for relaying the upload operation to the cloud storage (Algorithm 6) or for obtaining the token and signature of the file content. The proxying routine first verifies that the invoking user has write capabilities for the desired group (line 2). If positive, the content is authenticated (line 3) by using long term credentials PK_{TA}^{-1} provisioned during attestation. The file ciphertext and the corresponding signature are then uploaded to the cloud (line 4).

Users

Users *(i)* get their secret key from the AccessControl; *(ii)* write shared content; and *(iii)* read content shared with them. The write operation is shown in Algorithm 7. The writing user first randomly creates a file access key fk (line 2) and asks the AccessControl service $\mathcal{A}$ to envelope this key (line 3) for the group g_{id}, so that it can be anonymously deciphered by any member of it. He then encrypts the file using fk (line 4) and concatenates the two ciphertexts, key envelope and encrypted file (line 5), before transmitting the result to the WriterShield, so that it is uploaded to the cloud storage (line 6).

A-Sky satisfies the *lazy* revocation model [223], where a revoked user can continue accessing data created prior to revocation but not beyond that. Additionally, past data becomes inaccessible upon the first succeeding write to the same resource. The revocation is triggered by an administrator removing the user's id from the $group^r$ and $group^w$ access lists. Later, when new content is published in that group, a new random key is derived for encrypting the content and a new envelope is attached to it (Algorithm 7, lines 2-3). The revoked user's key will not be included in the envelope, and therefore he will be unable to access the new group key along with the newly published content.

Users read files according to Algorithm 8. First, they download the ciphertext package from the cloud storage (line 2), that can be validated by the signature check (line 3). In case the signature is valid, the user then splits the package between the key envelope and the file ciphertext (line 4). Next, the user iterates over user fragments in the envelope and tries to decrypt each of them with his secret key u_{sk} (lines 5-6). If successful, the obtained file access key can be used to symmetrically decrypt the file ciphertext (lines 7-8).

In case the indexed envelopes are used, the user read operation (Algorithm 9) requires the label reconstruction (line 5) followed by a binary search for the corresponding envelope fragment (line 6). Once located, the file access key is retrieved and the shared file is deciphered (lines 8-9).

As part of our prototype implementation, we developed a full-featured client in 1200 LoC of Kotlin. The client can be set up to operate in all possible configurations of A-Sky: keys in linear or indexed envelopes,

Algorithm 8: A-Sky: User read file.

```
input  : f: file reference
         u_sk: user secret key
output : P: plaintext file content
1  Procedure ReadFile(f, u_sk):
2      (C, σ) ← DownloadFromCloud(f)
3      if V(PK_TA, C, σ) = true then
4          envelope, cipher ← split(C)
5          forall pairs (k_c, t) in envelope do
6              fk ← AD(u_sk, k_c, t)
7              if fk ≠ ⊥ then
8                  P ← D(fk, cipher)
9                  return P
10     return ⊥
```

Algorithm 9: A-Sky: User read file with efficient decryption.

```
input  : f: file reference
         u_sk: user secret key
output : P: plaintext file content
1  Procedure ReadFile(f, u_sk):
2      (C, σ) ← DownloadFromCloud(f)
3      if V(PK_TA, C, σ) = true then
4          nonce, envelope, cipher ← split(C)
5          l_u ← H(u_sk || nonce)
6          (k_c, t) ← BinarySearch(l_u, envelope)
7          if (k_c, t) ≠ ⊥ then
8              fk ← AD(u_sk, k_c, t)
9              P ← D(fk, cipher)
10             return P
11     return ⊥
```

writes through the WriterShield, or through a standard proxy onto a Minio or Amazon S3 storage back-end with short-lived token-based authentication. Kotlin's full interoperability with the Java ecosystem allows us to easily integrate with the Java microbenchmark harness (JMH) [224] and Yahoo! cloud serving benchmark (YCSB) [225] frameworks that we use to perform the evaluation of A-Sky.

Evaluation

We evaluate the performance and scalability of our solution by first conducting micro-benchmarks. Then, we use the well-known YCSB [225] test suite to evaluate the overall system performance. All our experiments run on a cluster of 5 SGX-enabled Dell PowerEdge R330 servers, each having an Intel Xeon E3-1270 v6 processor and $64\,\mathrm{GiB}$ of memory. Additionally, we use 3 Dell PowerEdge R630 dual-socket servers, each equipped with 2 Intel Xeon E5-2683 v4 CPUs and $128\,\mathrm{GiB}$ of RAM. One of the latter machines is split in 3 virtual machines (VMs) to handle the roles of Kubernetes master, Minio server and benchmarking client (when a second client is needed). SGX machines use the latest available microcode revision `0x8e`, and have the Hyper-threading feature disabled to mitigate the Foreshadow security flaw [78].

Communication between machines is handled by a Gigabit Ethernet network. All our components can be independently replicated to provide availability, fault tolerance or cope with the load. Therefore, we have packaged our micro-services as individual containers, which we then orchestrate using an SGX-aware adaptation of Kubernetes [8]. When error bars are shown, they represent the $95\,\%$ confidence interval.

Cryptographic scheme

We first isolate and measure the performance of the underlying cryptographic primitive of A-Sky and compare it to the ANOBE scheme defined by Barth *et al.* [149] (*BBW*). Our implementation of *BBW* uses an elliptic curve integrated encryption scheme as the indistinguishability under non-adaptive and adaptive chosen ciphertext attack (IND-CCA)2 public key cryptosystem. Both key materials (*i.e.*, keys, curve) are chosen to meet $256\,\mathrm{bit}$ of *equivalent security strength* [226]. We also implement the efficient decryption of *BBW* as suggested in the paper by relying on the DH problem hardness, however in the context of elliptic-curve Diffie–Hellman (ECDH).

Table 4.3 shows throughputs for cryptographic key enveloping and de-enveloping considering that user keys are available. Apart from the experiment with efficient de-enveloping (where only one decryption is performed), units are in number of group members handled per second. While *BBW* can create envelopes at the rate of 330 members per second, A-Sky sustain rates 3 orders of magnitude bigger. Likewise, performance is improved by 2 orders of magnitude during de-enveloping operation. Such considerable difference is due to the performance gap between asymmetric and symmetric encryption primitives.

Table 4.3: Throughput comparison between A-Sky cryptographic scheme and *BBW*.

	Enveloping [$|\mathcal{G}|/\mathrm{s}$]	De-enveloping [$|\mathcal{G}|/\mathrm{s}$]	Enveloping Efficient [$|\mathcal{G}|/\mathrm{s}$]	De-enveloping Efficient [µs]
BBW	3.3×10^2	5×10^3	3×10^2	<4
A-Sky	1.9×10^6	2.5×10^6	1.2×10^6	<4
Ratio	5.8×10^3	5×10^2	4×10^3	*n/a*

$|\mathcal{G}|/\mathrm{s}$: group members per second

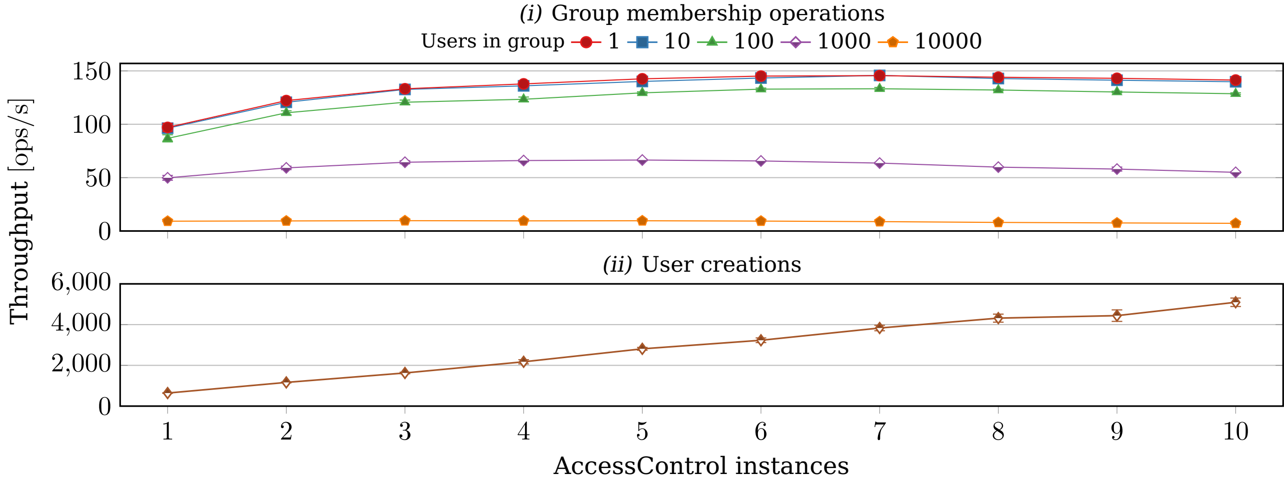

Figure 4.13: Throughput for administrative actions in the AccessControl: *(i)* adding or revoking users to/from groups of various sizes, and *(ii)* creating users.

BBW provides an *efficient decryption* mode that achieves less than 4 µs for the largest group size. However, this comes at the cost of a lower throughput during enveloping of 300 members per second, or 90% of the standard version. A-Sky is able to support the same decryption speed with indexed envelopes. In contrast, we achieve the rate of 1.2 million members per second when building envelopes (63% of standard version), three orders of magnitude faster than *BBW*. Furthermore, A-Sky produces ciphertexts of 60 B and 88 B per member for standard and efficient modes, respectively, whereas *BBW* produces 126 B and 154 B for the equivalent methods.

Scalability

In order to evaluate the throughput of operations performed by administrators when varying the number of AccessControl instances, we delegate to Kubernetes the distribution of requests among such instances, where a *service* is exposed. Figure 4.13 shows the results. The scalability of adding users to a group or revoking their access rights is limited, as these operations require one a read-modify-write (RMW) cycle to check and update the signature of the group *document*. The larger the group, the more time the operation takes as each signature encompasses every user within the group, *e.g.*, we reach an average of only 8.8 ops/s for groups with 10 000 users and 135 ops/s for groups containing a single user. This effect could be mitigated by, *e.g.*, batching together multiple operations on a given group. Diversely, the operation that creates users scales linearly with the number of AccessControl instances, allowing more than 5000 user creations per second with 10 replicas.

Next, we look at the number of keys that can be included in an envelope per unit of time, also when varying the number of instances of the AccessControl service. A close-to-linear trend can be observed according to number of instances in Figure 4.14, showing that this operation also benefits from horizontal scalability. For groups of 1000 to 10 000 members, the throughput ceases to increase with more than 7 instances as the MongoDB backend becomes a bottleneck. In smaller groups, the performance is diminished due to the overhead associated with each request (*e.g.*, network connection, REST formatting, enclave transitions),

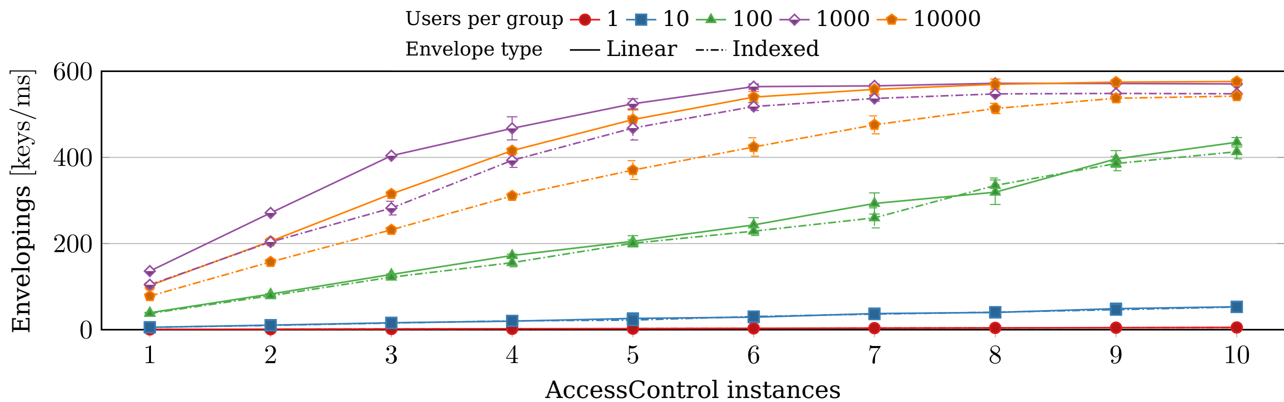

Figure 4.14: A-Sky: Throughput of enveloping a message for groups of various sizes with varying instances of the AccessControl micro-service.

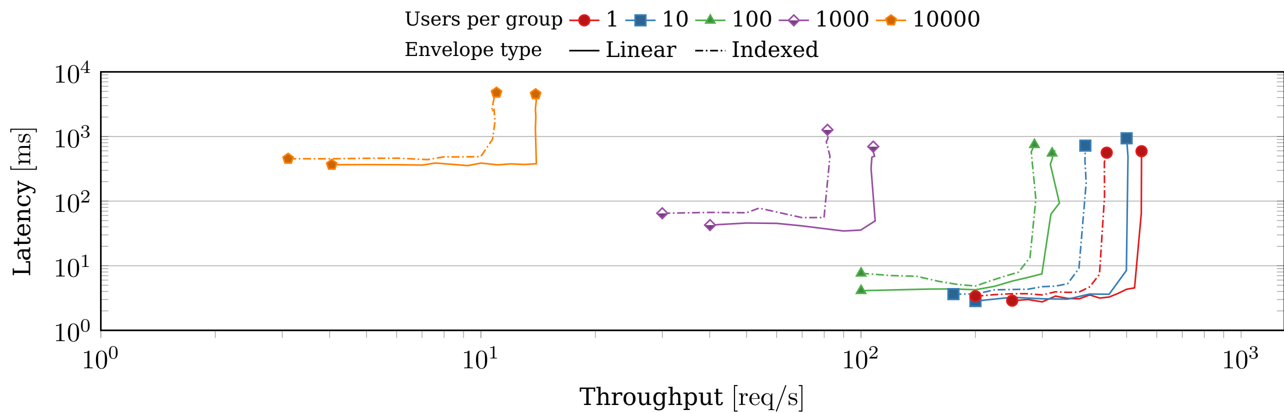

Figure 4.15: A-Sky: Throughput *vs.* latency of enveloping a message for groups of various sizes.

although it benefits from more AccessControl instances. Additionally, we ran the same experiment with the indexing feature turned on. For groups of 10 000 users, the throughput is reduced by 6 % to 26 %, having a marginal impact on smaller groups where the performance mostly depends on fixed costs.

We also evaluated the latency of the enveloping operation by increasing the throughput until saturation, again with indexing turned on and off. Looking at Figure 4.15, we notice that for groups which are larger than 100 users, latency increases linearly according to the group size, while the saturation throughput decreases linearly.

To evaluate the performance of the WriterShield, we conduct two experiments. In the first, data written to the cloud is proxied through the TEE, while in the other the WriterShield is solely used for generating temporary cloud storage access tokens, with write operations being proxied through a NGINX server in TCP reverse-proxy mode. In order to establish a baseline, we also wrote directly to the cloud storage service, without intermediaries. Results are shown in Figure 4.16. The bar plot on the left-hand side shows that for files of 1 kB and 10 kB the difference in performance is negligible, whereas bigger files cause more performance degradation when using the token feature. When the WriterShield is used to forward data instead (right-hand side), we see that the throughput increases with the number of service instances until it plateaus at about the same values as with the tokenized variant. For files of 1 MB, adding WriterShield instances renders no benefit. This effect happens due to the saturation of enclave resources acting as a TLS bridge between clients and the cloud storage server. Overall, using tokens would be the most efficient approach, although in this case the client would be responsible for using adequate proxies in order to hide its identity from the cloud storage.

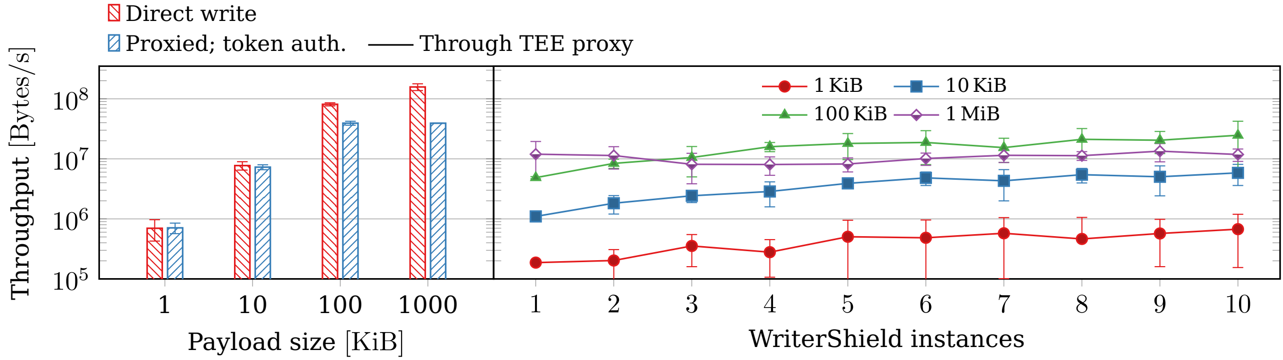

Figure 4.16: A-Sky: Throughput of writing data to the cloud storage in different ways: directly (baseline), through a TCP proxy using a temporary token for authentication, and through varying number of in-enclave WriterShield instances.

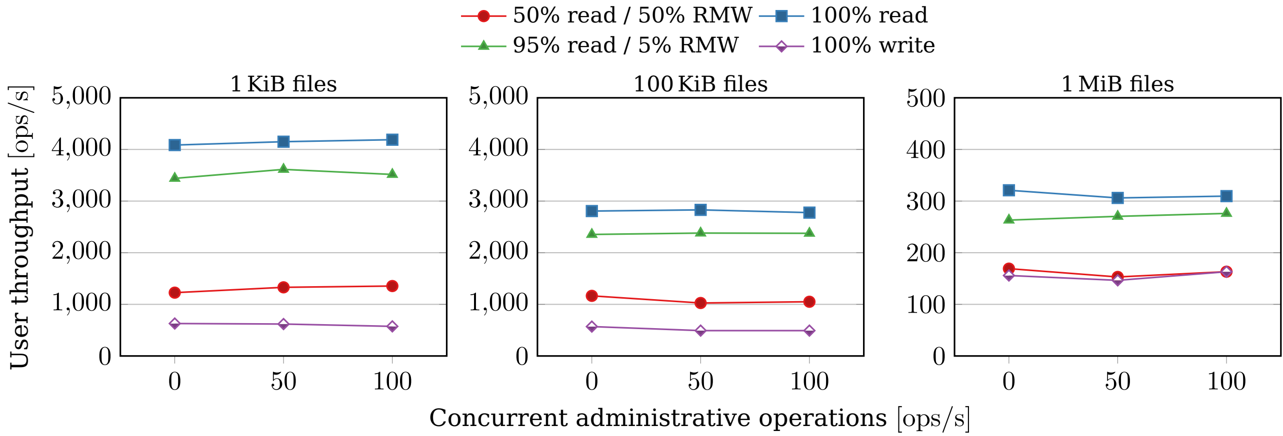

Figure 4.17: A-Sky: User throughput observed by our YCSB-based macro-benchmark, with various file access patterns, varying file sizes, and additioning simultaneous administrative operations.

Macro-benchmarks

To observe the behaviour of A-Sky under specific conditions of data serving systems, we use YCSB [225] workloads *A* (update heavy[1] 50/50), *B* (read heavy 95/5) and *C* (read only). Additionally, we include a write-only workload in our tests. As our system is not capable of direct-access writes, updates are replaced by RMW operations. We implemented an interface layer to link the benchmarking tool to A-Sky and ran each workload with 3 different file sizes: 1 KiB, 100 KiB and 1 MiB. We simulate 100 000 operations across 64 concurrent users and report the achieved throughput of user operations. Additionally, we activate a concurrent instance of YCSB that simulates 8 administrators doing group membership operations at three different rates: 0, 50 and 100 ops/s. The administrative operations are equally distributed between user additions and revocations, so that the user database size stays more-or-less constant.

Results are shown in Figure 4.17. Overall, we notice that user throughputs are not influenced by concurrent administrative operations, as each type of operation involves separate components of our architecture. For files of 1 KiB, an increasing proportion of writes causes a performance degradation from 4100 ops/s (read-only) down to 628 ops/s (write-only). For larger files (1 MiB), the difference is less dramatic, with throughputs going from 320 ops/s to 155 ops/s. Therefore, fixed costs are dominant when writing small files (*e.g.*, enveloping the file key), but are increasingly amortized for larger ones. We can also observe that the throughput in B/s (*i.e.*, multiplying the result in ops/s to the file size) is largely superior

[1] nomenclature used by YCSB: https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads

for larger files, as we have already noticed in Figure 4.16. In a nutshell, we retain that the end-user experience offered by A-Sky is not influenced by concurrent administrative operations, and that the overhead of the additional operations required for writing become smaller for larger files.

4.3 Summary

Diverging from intensive data processing protection within TEEs, this chapter rather focused on the generation and management of shared keys, which are obviously very sensitive information. More specifically, we were interested in schemes that allow users to share content in a secure manner. Besides guaranteeing data confidentiality and integrity, we also enforced access control, *i.e.*, being sure that only rightful users are able to read or modify shared content. The trusted environment brings new design perspectives regarding cryptographic protocols since shielded computation may operate on computationally simpler yet secure encryption constructs. We took advantage of this property by proposing and evaluating two original systems.

IBBE-SGX is a cryptographic access control extension that derives cuts in the computational complexity of IBBE thanks to secure enclaves. Such simplification came from the fact that the IBBE encryption operation took place in a TEE and could therefore use a master key that otherwise could not be available in untrusted environment. In such case, a much more complex operation involving a large public key would take place. Turning IBBE into a viable solution brought along its advantages regarding little metadata generation and the ability to dismiss the usage of a PKI.

Since we assumed that a variety of client devices should be able to access IBBE-SGX, possibly with no TEE available, the same encryption simplification could not be done for decryption. To mitigate the high performance cost, we proposed a group partitioning mechanism such that the complexity was bound to a fixed constant partition size rather than the size of the whole group. We conducted experiments with real and synthetic benchmarks, demonstrating that IBBE-SGX is efficient both in terms of computation and storage, even when processing large and dynamic workloads of membership operations. Apart from producing group metadata six orders of magnitude shorter than the traditional HE, which makes it more efficient both in terms of storage and bandwidth, our construction also performed membership changes at rates one order of magnitude faster than HE. At the same time, administrators cannot infer user keys and snoop into sensitive shared content.

Next, we introduced A-Sky, an end-to-end system that brings the additional guarantee of anonymity among group members. A-Sky leverages a TEE intermediary exclusively for the content sharing operation, while users can consume the shared content by reading it directly from cloud storage providers. We incorporated the cryptographic construction into a scalable system design that leverages micro-services that can possibly scale according to the undergoing access control and data sharing workloads. Results indicated that our cryptographic scheme is faster than state-of-the-art ANOBE schemes by 3 orders of magnitude and can serve groups of $10\,000$ users with a throughput of $100\,000$ key derivations per second per service instance.

Chapter 5

Privacy enforcement

Until now, we described the design of different kinds of services in order to protect data owned by their clients, whether they be at rest or being processed. These techniques are useful when one has access to the development stages of a system. However, sometimes users must rely on a service provided by third parties, who in turn may not be able or willing to change it. In such cases, privacy concerns may arise. Despite possible legal agreements, there is no fail-proof way of being sure that service providers do not stealthily track user habits or preferences while accordingly servicing their requests. The reasons for doing so can range from commercial (*e.g.*, targeted advertisements) to espionage (*e.g.*, by governments). To prevent that, we identify in this chapter ways of using trusted execution environments (TEEs) for privacy assurance.

We chose the domain of web search as it is the most prominent instance of such scenario. In Section 5.1.1, we describe a centralised proxy that obfuscates queries in order to make it hard for search engines to keep accurate user profiles. As any centralised approach, it suffers from the fact of simultaneously being a bottleneck and a single point of failure. In response to such disadvantage, we propose in Section 5.2.1 a decentralised peer-to-peer (P2P) solution that is better in terms of performance and fault tolerance and optimal in terms of accuracy.

5.1 Private web search

Web search is the most widely used online service, with billions of queries sent on a daily basis to Google alone. Indeed, search engines have become an essential service for finding content on the Internet. By regularly querying these services, though, users disclose large amounts of personal data (*e.g.*, [227, 228]). Queries are generally stored by search engines to analyse user behaviour and to personalize responses according to profiles inferred from past queries [229, 230]. Additionally, the economic model of many online services heavily relies on personalized advertising [231]. Numerous studies point, however, that the collection of search queries opens a number of privacy threats as they possibly disclose sensitive information about individuals (*e.g.*, age, gender, religion, political preferences, sexual orientation) [232].

To limit personal information disclosure, solutions enabling users to query search engines in a privacy-preserving manner have been proposed. They can be classified in categories, which enforce:

1. *unlinkability* between users and their queries.

   This is done by hiding identities of users through anonymous communication (*e.g.*, the onion router (Tor) [154], Dissent [157, 158] and RAC [156]). Systems in such category are limited for two reasons:

   - they typically suffer from poor performance because of the heavy cryptographic mechanisms on which they rely;
   - despite ensuring the requester's anonymity, it has been shown [161] that the actual content of search queries may be sufficient to link them back to the originating users' profiles by running *re-identification attacks* [162].

2. *indistinguishability* between user profiles and queries.

To that end, they obfuscate user profiles in such a way that the search engine cannot distinguish between users' real interests and fake ones (*e.g.*, Track me not [164], GooPIR [165]). These approaches generally operate by sending fake queries on behalf of the user.

It has been shown [162], however, that search engines may easily distinguish fake from real traffic due to external resources used for generating fake queries (*e.g.*, RDF Site Summary (RSS) feeds, dictionaries).

3. a combination between unlinkability and indistinguishability.

   The only existing solution of which we are aware, PEAS [166], assumes a weak adversarial model of non-colluding proxy servers.

4. *private information retrieval* (PIR) (*e.g.*, [233, 234]).

   These approaches build specialized search engines based on cryptographic techniques enabling to answer a user request without having access to its content. These techniques are, however, still unpractical due to their limited performance. Response times can reach seconds for very large data stores [153], which is the case of search engines.

Based on these considerations, it appears that in order to fully support privacy-preserving Web search one must provide a protocol that *(i)* enables the protection of requesters' identities in a realistic adversarial model; and *(ii)* provides effective indistinguishability with realistic fake queries, *i.e.*, difficult to distinguish from real ones.

5.1.1 X-Search

X-Search [20] consists of a proxy that enables Internet users to access Web search engines in a privacy-preserving manner. Instead of submitting queries directly to the search engine, a user sends them through an encrypted channel to the X-Search proxy, where the connection endpoint lies inside a software guard extensions (SGX) enclave. Once in there, queries are decrypted and manipulated in plain-text form before being forwarded to the search provider. Such manipulation involves the generation of an obfuscated query by aggregating k random past queries and the original one using the logical OR operator. As a consequence, the search engine is not able to distinguish which one is the original query. Due to the obfuscation, results returned by the search engine are mixed. The X-Search proxy therefore filters them and forwards only the results related to the initial query back to the user.

The X-Search protocol involves three entities:

1. the client, whose code and platform are trusted;
2. the X-Search proxy, which runs on cloud platforms and may behave in a Byzantine manner [235], *i.e.*, they are subject to failures, bugs or malicious behaviour; and
3. the search engine, which is honest-but-curious [236], *i.e.*, it correctly behaves when fetching answers to requests, while possibly collecting and exploiting the information they receive from clients. We also assume that search engines may run re-identification attacks (*e.g.*, [237]) in order to associate the received request to a known user profile. Besides, they may collude with proxy nodes (*e.g.*, Tor relays or X-Search proxies) in order to learn more about users.

Protocol overview

X-Search combines two complementary schemes: unlinkability and indistinguishability. The former hides the identity of requesting users; the latter their queries. Figure 5.1 depicts the architecture and execution flow of X-Search. The user interacts with the search engine through a X-Search proxy hosted on untrusted public cloud services and deployed on physical SGX-enabled nodes. As the proxy intermediates the contact between search engine and users, it hides their identities (*i.e.*, IP addresses). It is also in charge of obfuscating user queries and filtering the results returned from the search engine before forwarding them back to requesting users.

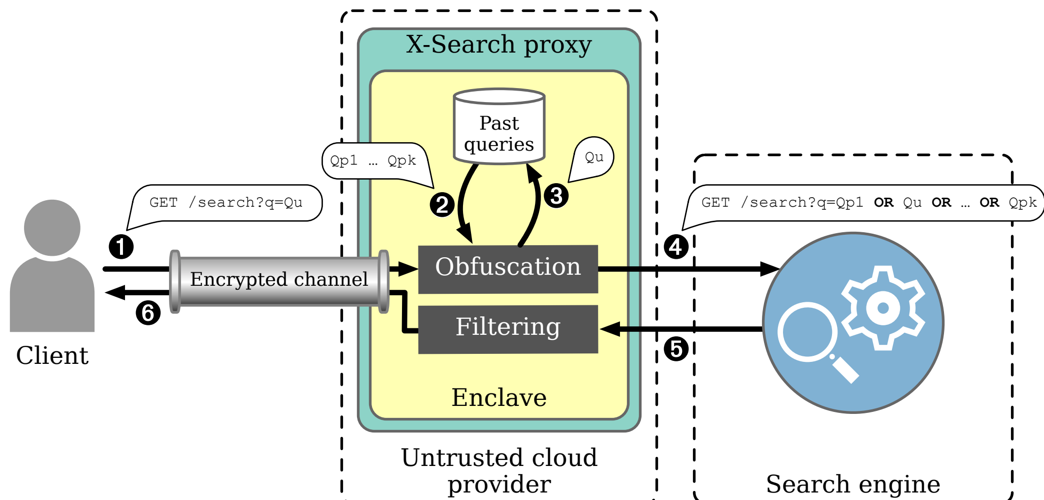

Figure 5.1: X-Search architecture and execution flow.

Initially, the user sends the query Q_u to the X-Search proxy (❶). The proxy then generates a new obfuscated query by retrieving k random past queries $Q_{p1}, ..., Q_{pk}$ (❷), which are aggregated to the original Q_u in a random order with the logical OR operator. Next, the proxy stores the initial query in the table of past queries (❸) and sends the obfuscated one to the search engine (❹). Unlike other indistinguishability protocols, X-Search reuses past queries as fake queries. Since they were sent by real users, they are effectively indistinguishable from the current user's real request. This is possible because past queries are securely stored inside the TEE with no correlation to the identity of their originating issuers. This database is therefore shielded against exploitation by malicious agents.

As the obfuscated query can alter the information returned by the search engine (❺) by mixing results for the original and fake queries, the proxy node performs a filtering step. It consists of removing all the results which are not associated to the original query. Finally, the remaining results are returned to the user (❻).

The X-Search proxy does not maintain individual profile structures associated to each user. Instead, it only updates a table containing the past queries. Moreover, the user sends queries to the proxy through an encrypted tunnel terminated inside the SGX enclave. Consequently, the protection of the original query is ensured from the client until inside the TEE of the proxy node. From the proxy to the search engine, the original query is protected thanks to the obfuscation mechanism.

Implementation details

The search unlinkability provided by X-Search relies on a query broker that runs within the client's domain. It consists of a local daemon process executing alongside the client's Web browser and is in charge of the SGX attestation step (see Section 2.2.3). When users issue a Web search query, their Web clients first connect to this local broker, which encrypts the request and forwards the cipher to an X-Search node hosted in an untrusted cloud provider. The X-Search proxy decrypts the cipher, generates the obfuscated query, and securely stores the original one in the SGX reserved memory. When the search engine sends back the response to the X-Search node, relevant results are extracted and delivered backwards to the broker. Finally, the broker decrypts the results and delivers it to the Web client.

To enforce indistinguishability, X-Search relies on an obfuscation mechanism that hides user queries among multiple fake ones (Algorithm 10), so that an adversary cannot distinguish them. More precisely, the original query is aggregated with k randomly chosen fake queries connected by logical OR operators (lines 1–8). Once the obfuscated query is generated, the initial query is stored in the history (line 9). These fake queries come from the table of past requests maintained in the private memory of the X-Search proxy (H). Due to the enclave page cache (EPC) limitation of SGX enclaves (see Section 2.2.2), we limit the size of H, so that a sliding window of the most recent queries are exploited. Since they are real, each sub-query of the obfuscated request can potentially be mapped to an existing user profile by an adversary

Algorithm 10: X-Search: Query obfuscation.

```
input : Q_u: initial query
        H = {Q_0, ..., Q_m}: query history
        k: number of fake queries
output: Q_obf: obfuscated query
1  Q_obf ← ∅
2  index ← random(k + 1)
3  while sizeof(Q_obf) <= k do
4      if index = 0 then
5          Q_obf ← OR(Q_obf, Q_u)
6      else
7          Q_obf ← OR(Q_obf, H[random(m)])
8      index ← index − 1
9  H ← H ∪ Q_u
10 return Q_obf
```

Algorithm 11: X-Search: Results filtering.

```
input : Q_u: initial query,
        fakes = {Q_p1, ..., Q_pk}: aggregated queries,
        R: set of results for Q_u ∨ Q_p1 ∨ ··· ∨ Q_pk.
output: R̄: filtered results
1  R̄ ← ∅
2  q+ ← Q_u ∪ fakes
3  for r ∈ R do
4      s_max ← −∞
5      for q_i ∈ q+ do
6          score[q_i] ← nbCommonWords(q_i, r)
7          s_max ← max(s_max, score[q_i])
8      if score[Q_u] = s_max then
9          R̄ ← R̄ ∪ {r}
10 return R̄
```

conducting re-identification attacks. The attacker would thereby introduce some noise in their source, which would make it less accurate and consequently hinder re-identification attempts.

In spite of that, the obfuscation mechanism has an impact on the results returned by the search engine. This is due to the fact that they contain a mix of answers corresponding to $(k+1)$ queries (*i.e.*, k fake queries and the real one). The X-Search proxy takes this into account by filtering the returned results, so that those which are not related to the initial query are removed (Algorithm 11). For each response r from the result set (line 3), the algorithm determines whether it corresponds to the initial query based on a *similarity score* assigned to each query q_i that composed the obfuscated request q^+ (line 5). Such score is calculated as the amount of common words (`nbCommonWords`) between the query q_i and the result r under evaluation (title and description—line 6). A result r is considered related to the initial query, and hence forwarded to the user, if the original query Q_u has the largest score s_{max} (lines 8–9).

To evaluate our system, we implemented in C++ a fully-functioning prototype based on Intel SGX software development kit (SDK) (v1.8) libraries and tools [42]. In order to avoid costly trusted/untrusted mode transitions, we limit the enclave interface to essential operations that deal with sensitive information (2 enclave calls (ecalls) and 4 outside calls (ocalls)). Concerning EPC memory consumption, an excessive amount could potentially be caused by the management of the past queries inside the enclave's protected memory. We evaluate this aspect later on.

Experimental setup and metrics

To assess X-Search, we use a real world Web search dataset from America online (AOL) query logs [167]. It contains approximately 21 million queries, issued by 650,000 unique users between March and May 2006. Our evaluation takes into account the 100 most active users (as in [166]), since they are the most exposed to an adversary that aims at discovering their identities. This is simply because they disclose more information to search engines, and therefore allow for meaningful profile histories. In order to build these user profiles, we split the dataset in training (two thirds of queries) and testing sets (one third).

We compare the robustness and quality of X-Search against two systems: Tor [154] and PEAS [166]; and a baseline solution, where users directly send their queries to the search engine without any protection. Tor provides unlinkability by leveraging several layers of encryption to hide user identities. PEAS, in turn, combines unlinkability—by relying on a trusted proxy— and indistinguishability— by obfuscating the original request with fake queries in random order. These fake queries are generated from the graph of co-occurrence between terms in the user history.

We assess X-Search along three dimensions: privacy (*i.e.*, the protection of users' data), accuracy (*i.e.*, the results' quality), and performance (*i.e.*, X-Search's efficiency in terms of throughput, latency and memory usage). To evaluate privacy, we leverage SimAttack [162] a re-identification attack that outperforms previous approaches. To run it, we assume the attacker holds a set of user profiles built at the learning stage. Next, X-Search intermediates the requests (testing dataset) between users and search engine. For each obfuscated query the attacker receives, it tries to identify both the requesting user and the original query among fake ones.

SimAttack is based on a similarity metric $s(q, P_u)$ that corresponds to the proximity between a query q and a user profile P_u. Such profile is assumed to have been built by the adversary before users started using any privacy protection. In our evaluation, P_u contains queries that belong to the training set and were issued by a user u. The metric accounts for the cosine similarity of q to all queries belonging to the user profile P_u and returns the exponential smoothing of these similarities. We empirically set the smoothing factor to 0.5 as this provided better re-identification rates. Considering one obfuscated request q^+, the function $s(q, P_u)$ is applied for every pair composed of the sub-query $q \in q^+$ and user profiles. A successful match happens when a single pair $\langle q, P_u \rangle$ achieves a score beyond a given threshold. Otherwise, the attack is considered as failed for the obfuscated query q^+ under scrutiny.

Once we obtain all successful matches, we assess privacy by establishing a metric called re-identification rate, which is defined as follow:

$$\textit{re-identification rate} = \frac{|Q_{id}|}{|Q|} \tag{5.1}$$

where Q_{id} is the set of correctly re-identified queries: both the initial query and the associated user; while Q is the set of original queries sent by users. This metric is defined between $[0, 1]$ where 0 represents the best solution (*i.e.*, no re-identification) and 1 represents the worse solution (*i.e.*, all queries are re-identified).

Since the obfuscation mechanism impacts the results returned by search engines, we evaluate the capacity of X-Search to filter responses not related to the initial query, *i.e.*, the results' accuracy. This is done by comparing the search engine's reply (first 20 results) for a non-obfuscated query and the X-Search's filtered results. For each value of k (*i.e.*, the number of fake queries), we pick a random subset of the testing workload composed of 100 queries. Our experiments use the Bing search engine[1]. At the time of our work, Bing supported only single words with the `OR` operator, albeit our dataset comprises multi-word real queries. To circumvent that, we simulated the answer to an obfuscated query by merging the result sets of each individual request.

The accuracy evaluation consists in comparing the lists of results associated to the original query R_{or} and the ones returned by X-Search R_{xs}, after obfuscation and filtering. We consider precision (*i.e.*, correctness) and recall (*i.e.*, completeness) as defined below. Both metrics are within the interval $[0, 1]$. The best accuracy is provided with precision and recall at 1.

$$\textit{precision} = \frac{|R_{or} \cap R_{xs}|}{|R_{xs}|} \qquad \textit{recall} = \frac{|R_{or} \cap R_{xs}|}{|R_{or}|} \tag{5.2}$$

To evaluate X-Search performance, we consider throughput, memory usage and round-trip time. The throughput (requests/second) allows the assessment of X-Search's scalability, since it indicates its capability to operate under adequate response times even with a growing number of requesting users. Memory usage measurement, in turn, allows for verifying when the EPC would become saturated, and therefore incur in larger overheads. Finally, we measured response times considering the complete chain, including the search engine delays, so that we could compare our proposal to alternative approaches.

[1] queries are directed to `http://www.bing.com/search=q?`

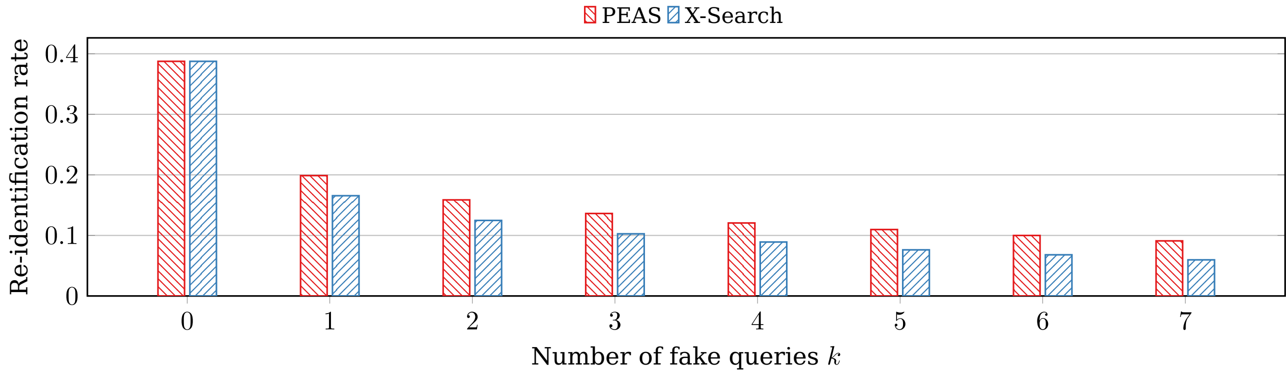

Figure 5.2: X-Search reduces the number of de-anonymized queries compared to PEAS.

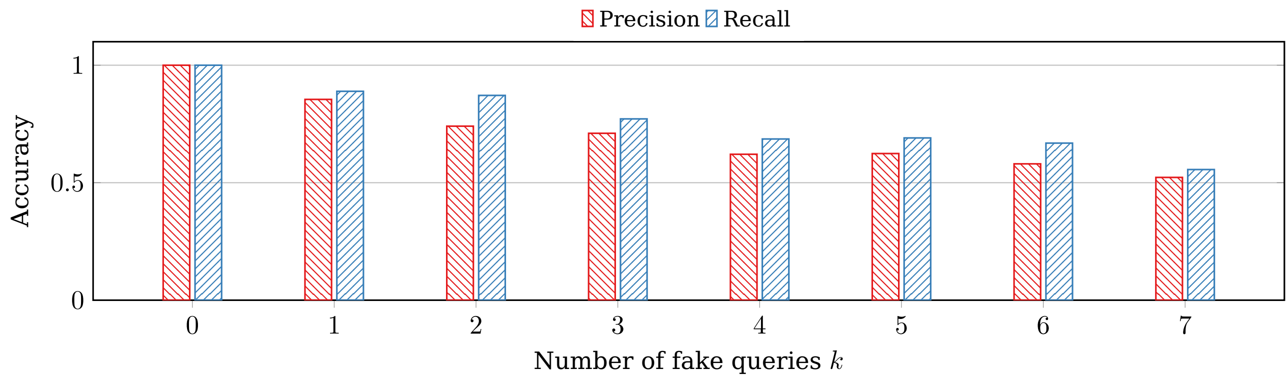

Figure 5.3: Results returned by X-Search are close to results associated to the original query.

Evaluation

X-Search was deployed on a machine with Intel® Core™ i7-6700 processor [196] and 8 GiB random-access memory (RAM) running on Ubuntu 14.04.1 long term support (LTS) (kernel 4.2.0-42-generic). First, we evaluate the capacity of X-Search to preserve user privacy and to improve user protection when compared to PEAS. To this end, we measure re-identification rates when leveraging SimAttack, as previously described. Figure 5.2 shows the results for PEAS and X-Search when varying the number of fake requests k. When $k = 0$, the evaluated systems enforce only unlinkability (*e.g.*, Tor). In such case, *i.e.*, without query obfuscation, the adversary is able to re-associate almost 40% of the queries to their originating user by making use of the profiles established with training data. This confirms that unlikability alone is not enough to effectively protect users against re-identification attacks.

Adding only one fake query drops this re-identification rate to $16\,\%$ for X-Search and almost $20\,\%$ for PEAS. This difference comes from the generation process of fake queries. Using real past queries makes X-Search more robust against re-identification attacks since all sub-queries can be mapped to past queries of other users, which creates confusion from the attacker's perspective. Contrarily, generating fake queries on the basis of co-occurrence of terms does not ensure PEAS to create the same kind of disorientation. The re-identification rate decreases according to the augmentation of k and X-Search provides better protection than PEAS in all scenarios. Such improvement varies from $23\,\%$ for $k = 1$ to $35\,\%$ for $k = 7$.

X-Search's accuracy is measured by quantifying its ability to remove results related to fake queries and keeping those resulted from the original query. Figure 5.3 depicts precision and recall according to raising values for k. As expected, both decrease as k increases, *i.e.*, more noise in the input causes more incorrect and incomplete outputs. Nevertheless, results returned to users are fairly accurate. For instance, the recall value for $k = 2$ is $87\,\%$, *i.e.*, only $13\,\%$ of the results delivered by X-Search are not part of the set returned by the search engine if the original query was sent directly. For the same value of k, precision

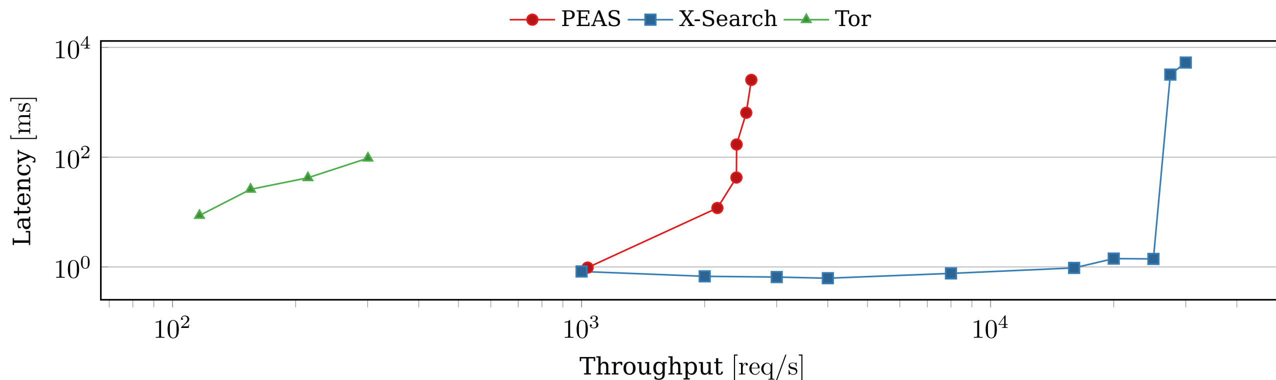

Figure 5.4: Latency vs. throughput comparison for X-Search proxy, PEAS and Tor.

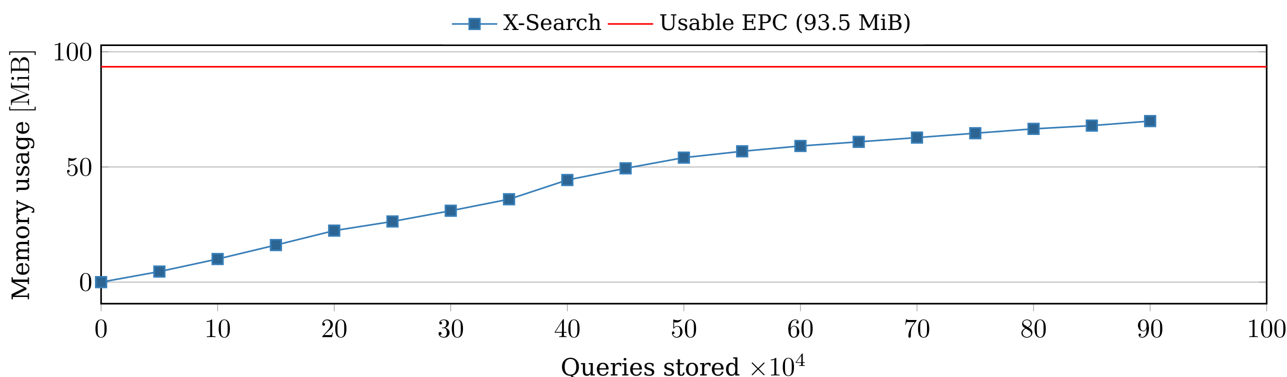

Figure 5.5: X-Search memory usage: The memory allowed to a single enclave can fit more than 1M queries before hitting the SGX EPC usable memory limit.

is equal to $74\,\%$, meaning that $26\,\%$ of the results returned to users are noise introduced by fake queries. These numbers confirm that X-Search preserves the quality of results.

With regard to system performance, we begin by observing the throughput $\times$ latency relation of the X-Search proxy. By iteratively increasing the rate at which requests are performed, we measure the latency to handle each request until it becomes too high. To do so, we use the wrk2 [238] workload generator without actually hitting the web search engine, so that we understand the saturation point of the SGX-based proxy in isolation. We compare X-Search against Tor and PEAS in similar conditions. From the usability perspective, unlike PEAS and Tor which require custom clients to forge messages that follow their protocol, X-Search can be used with third-party clients issuing regular hypertext transfer protocol (HTTP) requests, such as `wget` or `curl`.

Figure 5.4 presents these results. We observe that X-Search scales well, being capable of serving up to $25\,000\,\mathrm{req/s}$ with sub-second latencies. Differently, PEAS deteriorates much faster, with only $1000\,\mathrm{req/s}$ being served with sub-second latency. In our experiments, Tor performs very poorly: handling as few as $100\,\mathrm{req/s}$ at an average reply latency of $8.86\,\mathrm{s}$. This result confirms our implementation to be fast and scalable.

Next, we investigate how much memory is required by the obfuscation scheme. For this experiment, we used a larger dataset. Instead of considering only the 100 most active users, we use all the 6 millions unique queries available in the AOL dataset. To trace and profile the heap memory, we leverage Valgrind's Massif [239]. Figure 5.5 presents the results. Observing the trend of the X-Search curve, it is clear that the EPC size is largely sufficient to store at least 1 million queries, a number that can support with ease the obfuscation mechanism.

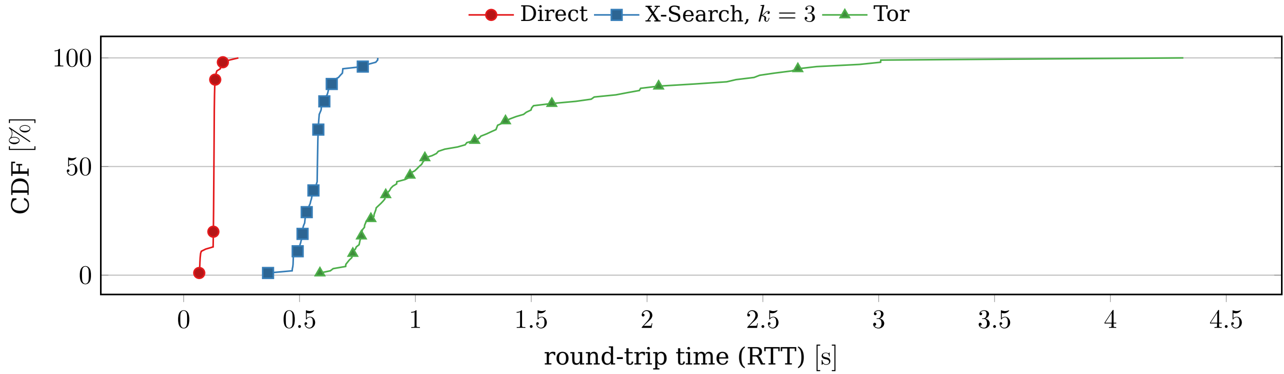

Figure 5.6: User-perceived web search round-trip time for 100 queries with X-Search, over the Tor network and directly contacting the web search engine.

Finally, we evaluate the end-to-end latency of a Web search query from submission to results' reception. Due to limiting policies adopted by Bing's search engine, we only issue 100 queries, picked at random in the AOL dataset. Our experiments were done in May 2017, and repeated for weeks in order to mitigate possible occasional disturbances in the service provider. We compare the observed round-trip time for three scenarios: *(i)* client directly contacts the search engine with no privacy guarantees; *(ii)* queries are routed through Tor network; and *(iii)* using X-Search. Figure 5.6 presents the results as a cumulative distribution function (CDF) of measured end-to-end latencies. We observe that X-Search allows for much faster replies in comparison to Tor. X-Search's median response time is $0.58\,\mathrm{s}$, and the 99th percentile is $0.87\,\mathrm{s}$. Tor, on the other hand, renders way worse results from the user perspective: the round-trip median time using onion routers was $1.06\,\mathrm{s}$, while the 99th percentile reached $3\,\mathrm{s}$. The Tor network largely exceeds well-known usability margins [240], while X-Search offers a usable yet secure browsing experience.

To summarize, we presented X-Search, a novel architecture to allow privacy-preserving Web searches that exploits Intel SGX to operate under stronger adversarial models than existing systems in literature. We contribute with a novel query obfuscation mechanism and the implementation choices of our full prototype. The assessment is made from three perspectives: privacy, accuracy, and performance. We analytically show that X-Search offers more vigorous privacy guarantees than its competitors as it operates under a stronger adversarial model. Furthermore, we experimentally demonstrate using a real dataset that X-Search is more resilient to a state-of-the-art re-identification attack than PEAS by $30\,\%$ in average. From the accuracy perspective, we show that the negative impact of X-Search's obfuscation scheme remains limited. Lastly, from the performance perspective, we show that X-Search outperforms previous privacy-preserving systems both in terms of latency and throughput.

5.2 TEE in the client-side

Although X-Search (see Section 5.1.1) advances private Web search in some aspects, centralised approaches like itself and PEAS [166] are not easily scalable and constitute a single point of attack and failure. For instance, they may be easily blocked by search engines that have anti-bot policies. Besides, results filtering due to query obfuscation are not perfect and cause lower quality responses. Moreover, in previous approaches, all queries are treated equally, *i.e.*, with the same protection level. Taking query obfuscation as an example, all requests are appended with the same amount of fake queries. This is not always effective as user searches may have different levels of sensitivity, *i.e.*, non-sensitive queries may be overprotected, while the opposite may happen to sensitive ones. We propose, in this section, a solution that tackles these issues: privacy, accuracy, scalability and employs adaptive protection.

The scalability aspect, in particular, is of special interest. Up to now, all proposed solutions could be possibly deployed in a third-party service provider. Contrarily, we now shift this design aspect to the client-side. We assume that each client node supports SGX, as it has been shipped in Intel processors targeting desktops and portable computers since 2015, therefore being by now fairly widespread. By

doing so, general user machines—which are more vulnerable to generic malwares and simpler infection vectors like social engineering—may be regarded as trusted. This brings new possibilities like server processing offloading [168], network monitoring [241] or trusted decentralised P2P networks, as we propose here.

5.2.1 Cyclosa

We present Cyclosa [21], a decentralised private Web search solution that leverages TEEs and:

(i) enforces *privacy* by protecting users against re-identification attacks through unlinkability and indistinguishability;

(ii) enforces *accuracy* by providing users with similar responses to those they would get by directly querying the search engine; and

(iii) is *scalable* to millions of users while enforcing service availability even if search engines limit bulk requests.

To enforce unlinkability between a query and its sender, each node participating in Cyclosa acts both as a client when sending its own requests and as a proxy by forwarding queries on behalf of other nodes. Each node has a TEE such as SGX wherein Cyclosa keeps secure inter-enclave communication endpoints and from where interactions with search engines are triggered. These nodes are therefore regarded as *untrusted* from the other peers' perspective and therefore no query information is leaked to them. The enclave they host, on the other hand, is trustworthy.

Re-identification attacks are mitigated by sending both fake queries and the real user query through multiple paths to the search engine. Differently from X-Search though, instead of blindly sending the same amount of fake queries regardless of the real query, Cyclosa assess its sensitivity and adapts the amount of noise to be sent along. This query sensitivity evaluation is made by quantifying two attributes:

(i) *linkability*, which relates to the current request's similarity with the user local profile: the more similar the bigger the chances of user re-identification; and

(ii) *semantic sensitivity*, based on the query's topic. Out of a predefined set of topics, users pick a subset that they consider to be sensitive.

Whenever a query is sent to the search engine, Cyclosa checks whether it is linkable to its issuer profile and if the query's topic belongs to user-defined sensitive topics. Based on this, it accordingly adjusts the amount of fake queries, *i.e.*, Cyclosa strongly protects sensitive queries and avoids overloading the system with fake ones when they are not sensitive.

In order to provide accurate results for clients, Cyclosa sends the real and fake queries through distinct relays. In this way, filtering becomes straightforward, since results that correspond to fake requests can be entirely discarded. Therefore, Cyclosa returns the same responses as when users directly query the search engine, hence reaching perfect accuracy.

Cyclosa's decentralised architecture consists of nodes with equal roles. This allows the system to scale well, even for a large number of clients as the load gets evenly distributed between participating nodes. Moreover, it brings advantages regarding fault tolerance: as there is no central point of failure, the system remains operative even in presence of localised attacks or blacklisting by search engines.

We implemented and evaluated Cyclosa using the same workload we used previously: the query logs extracted from the AOL dataset [167]. Results show that Cyclosa meets the expectations concerning load balancing, economy of resources and privacy protection.

Adversary model

The computations performed by Cyclosa for each user query go through three premises, namely:

(i) the **client machine**, which is trusted by its owner but untrusted by peers that compose Cyclosa P2P network. Such trust includes all computations that involve client information (*e.g.*, assessing the sensitivity degree of a query), which are performed locally outside of enclaves. Furthermore, we

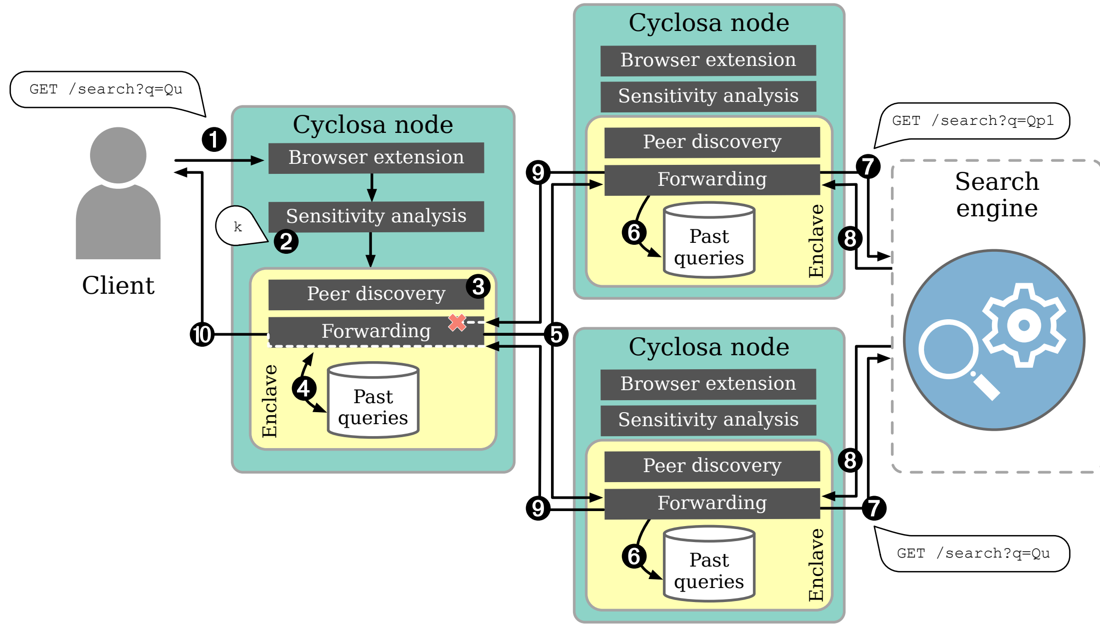

Figure 5.7: Cyclosa architecture and operating flow.

assume that the interaction between a human user and Cyclosa is trusted, *i.e.*, an adversary cannot modify a typed query, nor it can tamper with the local configuration of Cyclosa that a human user has set up, that is, the set of topics the user considers as semantically sensitive.

Clients cannot bypass the SGX enclave. Session keys are generated between pairs of Cyclosa enclaves after successful remote attestation. Therefore, untrusted code is not able to forge requests correctly encrypted and signed. If the client is breached though, the sensitivity analysis could be subverted. As third-party sensitive data (*e.g.*, table of past queries) is only handled inside enclaves, such attack would only compromise the infected peer;

(ii) a set of **proxies**, *i.e.*, forwarding peers who mutually distrust each other. They can act in a Byzantine manner [242, 235], *i.e.*, they can arbitrarily crash, be subject to bugs or under control of malicious adversaries.

Inter-enclave traffic as well as between enclaves and the search engine are protected by encrypted channels. In addition, query forwarding performed by the proxy is done inside the SGX enclave. Consequently, a malicious or compromised proxy cannot hamper Cyclosa (side-channel attacks are not considered - see Section 2.2.6). However, malicious hosts may replay past queries. This threat can be mitigated by including a random identifier in each message to detect replays of a limited set of recent messages. Also, a malicious proxy can deny initialisation or calls into enclaves. Cyclosa tackles this issue by letting clients blacklist non-responding or slow peers; and

(iii) the **search engine**, which is honest but curious, *i.e.*, it faithfully replies to search queries while possibly gathering information to build user profiles and run re-identification attacks [162].

The search engine's capability of re-identifying users is very limited (more in the evaluation section). However, the search engine could identify a real query when it receives this query for the first time. Later on, the request tends to become indistinguishable from others as it will be re-issued as fake. Even in this case, the identity of the requesting user is still hidden by the proxy (unlinkability).

Protocol overview

To use our system, clients install the Cyclosa browser extension, so that they seamlessly get protection without changing their browsing habits. Figure 5.7 illustrates the operating flow. When the user formulates a query Q_u (❶), Cyclosa evaluates its sensitivity (❷) by checking whether the query is linkable to the user profile and if there is a match between the semantic analysis of Q_u with the user-defined set of

sensitive topics previously established. As a result, a score k is produced and used as the number of fake queries needed to make the real user query indistinguishable from others.

A peer discovery component (❸) then selects $k+1$ random peers $P_{p0}, P_{p1}, ..., P_{pk}$ to which it sends (❺) the real query Q_u and k fake queries $Q_{p1}, ..., Q_{pk}$ obtained from the database of past queries (❹), issued by other users in the system. They are stored in a local table whenever a node acts as a relay. As indicated in X-Search evaluation, real past queries used as fake make them seem real from the search engine's perspective. In contrast, systems such as TrackMeNot or GooPIR, where fake queries are generated using RSS feeds or dictionaries, are more easily recognized as artificial.

When a request is received by peers acting as relays, it is stored in the local table of past queries (❻) before it is forwarded to the search engine (❼). Relays are not aware of whether the queries they handle are real or fake. An external observer therefore learns nothing by analysing the encrypted network traffic. In contrast, systems where fake queries are appended in the relays (*e.g.*, X-Search or PEAS) allow an adversary to infer whether an outgoing message corresponds to a real query or an obfuscated one based on the request size, since obfuscated queries built with the `OR` operator are larger in length.

Upon receiving a request from the node acting as a relay, the search engine provides it with the query results (❽). Such responses are then routed back to the original client (❾). Finally, Cyclosa drops the responses corresponding to fake queries before the user gets access to the initial query's results (❿).

To avoid information leakage, all Cyclosa components that process sensitive data, *i.e.*, queries issued by other users, are located within enclaves. In order to minimize the amount of trusted code, however, components that process data related to the user who owns the machine are placed outside the enclave. Specifically, the unit in charge of query sensitivity assessment issued by the local user is performed outside the enclave. This allows to drastically minimise the amount of trusted code, which reduces the risk of having critical bugs and unnoticed vulnerabilities. Since remote peers are untrusted, components that handle other users' queries in plain text are placed within enclaves, along with the communication endpoints where they are encrypted and decrypted before or after transmissions. Components placed inside trusted environments include: peer discovery, query forwarding, encrypted communication endpoints and the past user queries database.

Query sensitivity analysis

Aiming at avoiding the network overload potentially caused by excessive noise injected by our proposal yet achieving indistinguishability of sensitive queries, Cyclosa performs a sensitivity analysis. Specifically, it dynamically protects user queries according to their susceptibility by adjusting the amount of fake queries sent along with them. To quantify query sensitivity, Cyclosa relies on two measurements computed outside the enclave:

(i) semantic assessment, *i.e.*, indicates whether a query is related to sensitive topics previously declared by the user; and

(ii) linkability assessment, *i.e.*, the risk that the query gets linked back to its issuer by means of comparison with past queries issued by such user.

Since semantic sensitivity is subjective, *i.e.*, one query might be considered as sensitive by one user and non-sensitive by another, Cyclosa proposes a user-centric approach. Users select what they consider semantically-sensitive topics among health, politics, sex, and religion[2]. Nevertheless, a user can import dictionaries to create other sensitive topics. The **semantic-based** assessment's result is binary and indicates whether the query belongs to at least one topic marked as sensitive. To achieve that, we use information retrieval approaches and build dictionaries of terms associated to each sensitive topic.

These dictionaries are built with keywords based on two datasets:

(i) WordNet [244], a machine-readable lexical database that is organised by meanings, where words are grouped into sets of synonyms called *synsets*; and

[2] Based on Google's privacy policy [243] which defines sensitive personal information as *"a particular category of personal information relating to topics such as confidential medical facts, racial or ethnic origins, political or religious beliefs, or sexuality."*

(ii) eXtended WordNet Domains [245], which maps WordNet synsets to 170 domain labels. By using this, we identify keywords related to our privacy-sensitive topics.

Additionally, we leverage a statistical approach to enrich our keyword dictionaries of sensitive topics. It consists of latent topic models, or latent Dirichlet allocation (LDA) [246], a generative probabilistic model which captures correlations among words and topics that is well adapted for modelling text corpora. In the LDA model, a topic is described through thematic vectors that indicate the topic's latent dimensions. Once this model is trained with a text corpora associated to the user-defined sensitive topics, we build the dictionary by gathering the terms of all thematic vectors.

In the experiments, we consider sexuality as an example of sensitive subject in user queries. We trained a LDA statistical model using the Mallet toolkit [247], with 200 topics on 2 million titles and descriptions of videos related to the sensitive subject [248]. Finally, a query is identified as semantically sensitive either if it contains a term in at least one LDA topic or if it is linked to a WordNet domain tagged as sensitive.

The **linkability** assessment's goal is to determine if the query is vulnerable to a re-identification attack. In such attacks, an adversary tries to link an anonymous query to a specific user by measuring the distance between the query and a set of user profiles built from past queries that were collected, for instance, before users started using private Web search mechanisms. Concretely, the linkability assessment performed on the client side provides a score l within the interval $[0, 1]$ that corresponds to the current query's proximity to the user profile. To do that, we first represent the user query in a vector where each element is a term in the query. Then, the cosine similarity between this vector and the one corresponding to each past query made by the user is computed. Finally, we use exponential smoothing as a window function[3] to produce an aggregated score. Instead of ordering the samples by time though, we order them by cosine similarities to compute this score, so that the similarity has more importance than how long ago a given query was submitted.

The number of fake queries k may be defined based on one or both assessments. As a result from the semantic evaluation, we simply obtain a boolean that indicates whether the current query is related to sensitive topics. If it is the case, k is defined to a maximum value k_{max} and the linkability assessment can be skipped. Otherwise, *i.e.*, when the query is not semantically sensitive, k is defined as the linear projection of the linkability score l with regard to k_{max}, *i.e.*, $k = l \cdot k_{max}$, with $l \in [0, 1]$.

Enclave operation and implementation details

Once the number of fake queries k is decided according to the user query sensitivity, the process continues in the SGX enclave as it involves forwarding sensitive data to remote peers. Each node discovers these peers and dynamically maintains a view of other alive nodes in the system. When a query is issued, Cyclosa randomly picks $k + 1$ out of these nodes to act as proxies for the original query and k fake ones—randomly selected from the table of past queries. Inter-enclave communication is always encrypted and the identity of the proxy handling the original query is kept in a table within the enclave.

Once a proxy receives a query forwarding request, it adds this query in its local table of past queries and routes it to the search engine. Answers from the search engine are returned to the original user who, in turn, receives the result that came from the proxy which carried the real query, silently dropping all others. Routing real and fake queries through different paths makes them indistinguishable from the search engine's perspective.

At bootstrap phase, besides the user declaration of sensitive topics, there are other key elements that need to be initialised when Cyclosa is first launched by a given user:

1. Initially, there are no past requests stored in the enclave. Hence, Cyclosa fills the table with popular Google queries [249], as they are issued by real users regarding trendy topics.
2. Then, peer discovery is bootstrapped. We assume it is done as in classical P2P systems using public repositories of IP addresses (*e.g.*, Tor) from which a Cyclosa instance can select a first sample of random peers. Peer discovery is done using classical algorithms and contributions to this field fall

[3]Window functions consider only a limited interval of the samples to compute a final score, usually based on time, *i.e.*, the most recent samples. Exponential smoothing, unlike the moving average where samples are equally weighted, assigns exponentially decreasing weights to "*old*" samples.

outside the scope of this work. For implementation purposes of our prototype, we use Zyre [250], a library based on ZeroMQ [172] for peer discovery, groups organisation and multicast of events.

3. Finally, SGX remote attestation of connecting peers must be performed (see Section 2.2.3).

To allow seamless integration to end-users' workflow, Cyclosa was designed as an extension to the Firefox browser. It is JavaScript-based and integrates the SGX enclave using *js-ctypes*, which allows asynchronous calls between the enclave and the untrusted extension code. Connections to search engines, in turn, are established with transport layer security (TLS) tunnels. They must originate from enclaves in order to prevent sensitive data disclosure. To do so, we use a SGX-compatible version of mbedTLS [251]. All in all, the Cyclosa enclave object accounts for only 1.7 MiB. This and a circular buffer for the past queries table prevents Cyclosa from suffering of EPC paging.

Experimental setup and metrics

We compare Cyclosa against five approaches for protecting privacy in Web search: Tor [154], TrackMeNot [164], GooPIR [165], PEAS [166] and X-Search [20]; and a protection-free Web search scenario (see Section 2.6). As in Section 5.1.1, we use the AOL query log dataset [167] and the same methodology described in [237] to evaluate privacy by considering a subset of the most active users. Here, however, we manually extracted 198 users who sent at least one semantically sensitive query. Again, to perform a re-identification attack, we split the queries into training set (two thirds) and testing set (one third). In the training set, this represents an average of 487.6 queries per user out of 96.547 queries.

To determine user-perceived sensitivity regarding Web queries, we conducted a crowd-sourcing campaign using Crowdflower [252]. We selected the first 10,000 queries over all user queries in the testing set, and asked the workers to determine if these queries are related to sensitive topics (*e.g.*, health, politics, religion and sexuality). Each query was annotated by 5 distinct workers. It resulted that only 15.74 % of the queries are related to sensitive topics. Since the great majority are non-sensitive, lots of resources (*e.g.*, processing, storage, bandwidth) would be wasted should all queries be equally protected as in X-Search. This motivates Cyclosa's adaptive approach, which applies a dynamic protection scheme to sensitive queries.

To measure the accuracy of Cyclosa's ability to automatically determine if a query belongs to a sensitive topic, we consider the precision and recall metrics. The *precision* indicates the proportion of truly sensitive queries among those detected as such by Cyclosa (correctness), while the *recall* express the ratio between the queries detected as sensitive with regard to all sensitive queries in the testing set (completeness). We consider the crowd-sourcing annotations as ground truth. Let Q_s be the set of actually sensitive queries (*i.e.*, related to sensitive topics), and Q_m the set of queries that are identified as sensitive by Cyclosa. Precision and recall are defined as follows:

$$precision = \frac{|Q_m \cap Q_s|}{|Q_m|} \qquad recall = \frac{|Q_m \cap Q_s|}{|Q_s|}$$

To evaluate privacy, we use SimAttack [162] in order to match queries with user profiles (see Section 5.1.1). Then, we calculate the overall re-identification success rate according to Equation (5.1), *i.e.*, the proportion of queries for which the user profile is successfully re-identified with respect to all queries in the testing set. Naturally, the lower the re-identification success rate is, the better is the privacy level.

By design, Cyclosa returns to users the same results associated to their search query as if no privacy-protection mechanism were employed. However, other proposals such as X-Search and PEAS include an obfuscation scheme that impacts these results by adding some noise to what is returned by search engines. Because of this, we evaluate the capacity of such approaches of returning results related to the initial query. To achieve that, we measure accuracy in terms of precision and recall according to Equation (5.2), *i.e.*, by comparing results returned by direct requests to search engines and those resulted from using privacy-preserving strategies.

To evaluate the behaviour of Cyclosa from a systems perspective, we measure the end-to-end delays and throughput versus latency. Besides, we observe the impact of the amount of fake queries on latency and

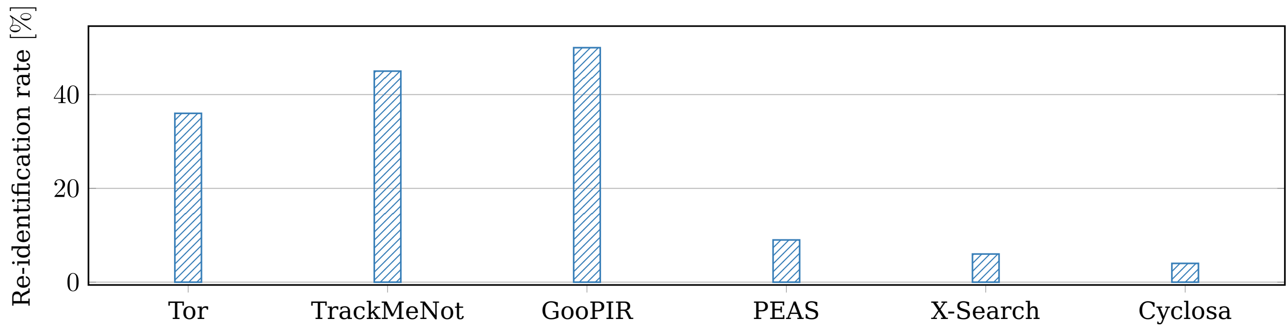

Figure 5.8: Comparison of Cyclosa's privacy level with other approaches ($k = 7$ when they employ obfuscation).

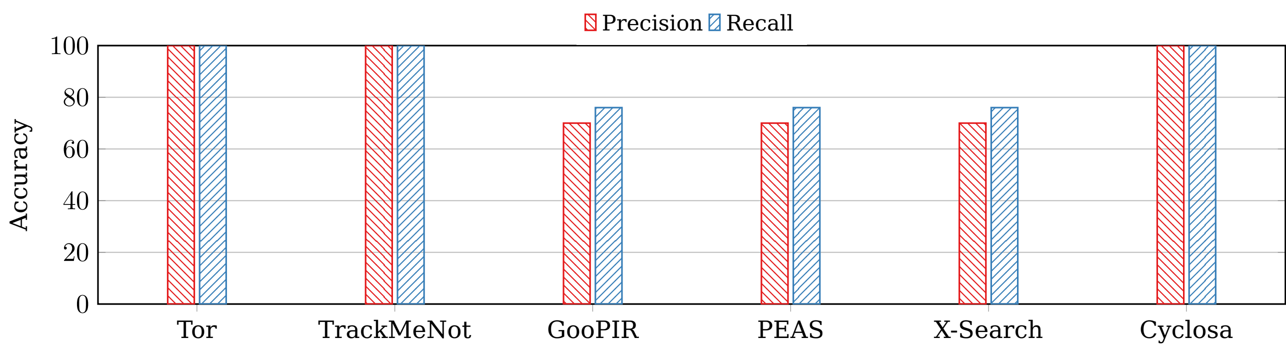

Figure 5.9: Accuracy of results returned by Cyclosa and alternatives ($k = 3$ when obfuscation is employed).

Cyclosa's ability to avoid being blocked by search engine's anti-bot prevention mechanisms in comparison to X-Search.

Evaluation

We first evaluate the capacity of Cyclosa to protect user privacy by measuring its robustness against an adversary conducting a re-identification attack. Figure 5.8 shows re-identification rates for Cyclosa, Tor, TrackMeNot, GooPIR, PEAS, and X-Search with $k = 7$.

Without query obfuscation (*i.e.*, Tor), an adversary is sure that every query has been issued by a user. Consequently, the challenge in this case consists of mapping each query to user profiles built from previously collected user information. Our results show that an adversary is able to re-affiliate around $36\,\%$ of the new queries to their correct original users. Interesting enough, the re-identification rate for Tor is the same of PEAS, X-Search and Cyclosa with $k = 0$ (not shown in the plot).

Without unlinkability (*i.e.*, TrackMeNot and GooPIR), the adversary needs to distinguish real queries from the fake ones. Results show that a large proportion of real queries are identified: 45% and 50% for TrackMeNot and GooPIR, respectively. This high re-identification rate is mainly caused by the fake query generation process, which uses RSS feeds. When sources for fakes are far from the users' interests, the adversary can easily dissociate them.

Combining query obfuscation and unlinkability drastically drops re-identification rates. Indeed, the adversary's challenge becomes harder. It consists of retrieving both the user's identity and identifying real queries among fake ones. By using real past queries as fake ones, X-Search and Cyclosa provide lower re-identification rates than PEAS, which uses a graph of co-occurrence of terms built from past queries. That is, terms that happen to be associated more often in real queries will compose the fake query along with the one issued by the user.

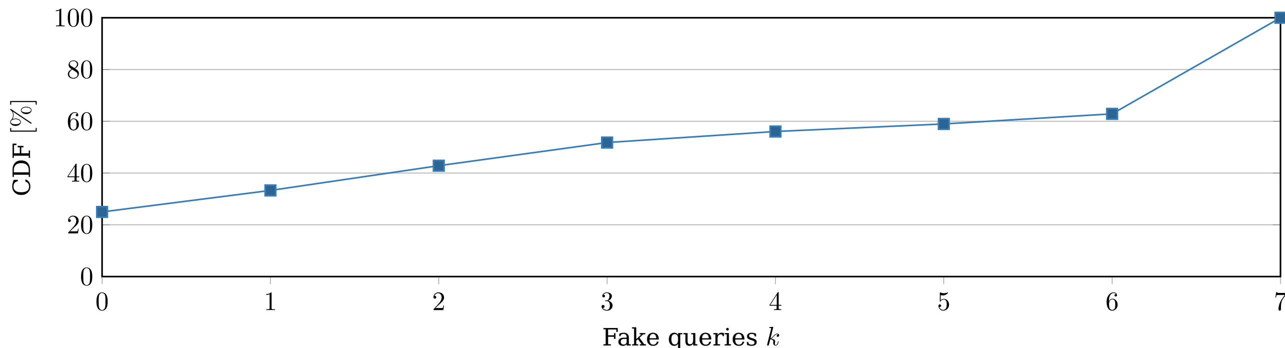

Figure 5.10: Actual number of fake queries in Cyclosa.

Table 5.1: Detection of semantically sensitive queries.

Semantic tool	Precision	Recall
WordNet	0.53	0.83
LDA	0.84	0.89
WordNet + LDA	0.86	0.85

Moreover, Cyclosa slightly reduces this re-identification rate in comparison to X-Search from $6\,\%$ to $4\,\%$. This difference comes from the obfuscation scheme adopted by them. For X-Search, the adversary receives a group of $k+1$ queries and has to identify the real one in this group. In Cyclosa, however, the adversary receives individual queries from different proxies and has to decide whether it is real or fake. Due to the increased amount of confusion it creates, the re-identification becomes harder.

Next, we evaluate Cyclosa's accuracy, *i.e.*, its ability to return the same answers from the search engine as the ones resulted from unprotected direct queries. Figure 5.9 shows the obtained accuracy in terms of correctness and completeness of answers returned by Cyclosa and alternatives, with $k = 3$ when query obfuscation is used. There are two sets of solutions. Cyclosa and TrackMeNot provide perfect accuracy as they differentially handle real and fake queries' responses, while Tor achieves the same because it does not employ obfuscation. The other solutions provide lower accuracy because, to some extent, they are not able to distinguish between answers of fake and real queries. Precision for GooPIR, PEAS and X-Search reaches 65% for a recall of 70%, when $k = 3$ and decrease for larger values of k.

Cyclosa dynamically and adaptively protects queries according to their sensitivity by adjusting the amount of noise. Figure 5.10 shows the CDF of the number of fake queries induced by Cyclosa in our testing set when the maximum value for k is 7. Results show that $25\,\%$ of queries do not need obfuscation, and $50\,\%$ of them use less than 3 fake queries. The sharp increase for $k = 7$ corresponds to requests identified as highly sensitive in the semantic-based assessment, and consequently scaled to the maximum protection level. They amount to $35\,\%$ of our testing set. In contrast, X-Search would have generated this maximum number of fake queries for all queries.

As previously defined, the sensitivity of a query is evaluated in two dimensions: linkability to its issuer's profile and as to whether it belongs to user-defined sensitive topics. This detection is based on both WordNet libraries and a trained LDA statistical model. To evaluate their influence on the results' quality, we ran them individually and combined while obtaining precision and recall when detecting whether queries belonged to the sensitive topic related to sexuality. Table 5.1 shows the results. Overall, most of queries are detected, with a recall between $83\,\%$ for WordNet and $89\,\%$ for LDA. Precision varies from $53\,\%$ (WordNet) to $86\,\%$ (WordNet + LDA). In our experiments, combining WordNet and LDA provides the best trade-off between correctness and completeness.

To analyse Cyclosa's performance, we start by showing in Figure 5.11 the observed end-to-end latency in comparison to X-Search and Tor, including the time it takes to the search engine for processing the requests. We further include the measurements achieved without any protection by directly contacting

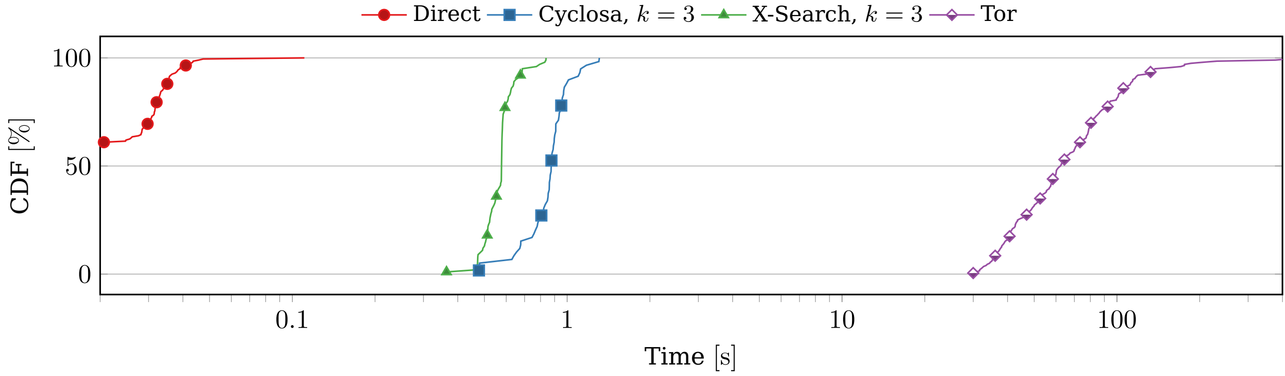

Figure 5.11: Distribution of end-to-end delays for 200 queries, $k = 3$.

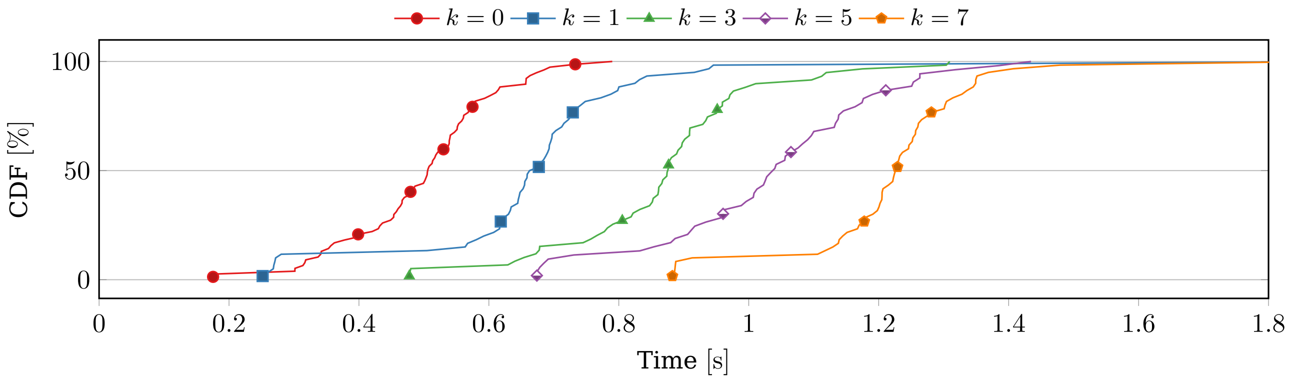

Figure 5.12: Cyclosa: Impact of k on latency.

the search engine. As expected, Tor is the slowest, with a median latency of $62\,\mathrm{s}$. Both Cyclosa and X-Search allow for sub-second delays for the large majority of the queries, with medians of $0.88\,\mathrm{s}$ and $0.58\,\mathrm{s}$, respectively.

In Figure 5.12, we explore the impact of changing the number of fake queries. Even in the worst case ($k = 7$), the system still returns the results to clients in less than $1.5\,\mathrm{s}$, with median latency at $1.2\,\mathrm{s}$, which still allows for a usable web browsing experience.

We also evaluate the capacity of a Cyclosa node to sustain high rates of requests in comparison to X-Search. Figure 5.13 presents the results. We submit requests at increasingly high constant rates and measure the latency of a reply from the next hop (the X-Search proxy or a Cyclosa relay), but without actually submitting the requests to the search engine. Cyclosa is able to handle very high request rates with sub-seconds response delays. In our evaluation, we achieved $40.000\,\mathrm{req/s}$ with $0.23\,\mathrm{s}$ median response time while X-Search starts straggling at $30.000\,\mathrm{req/s}$.

Such rates, although possible, cannot be observed in practice without being immediately blocked by a search engine's malicious user detection system. In our experiments with Google search engine, this happened very soon. Moreover, users submit searches at rates that are orders of magnitude slower: the 100 most active users from the AOL dataset perform queries at a rate of $31.23\,\mathrm{req/h}$. Considering the same queries, X-Search induces $10.500\,\mathrm{req/h}$ among real and fake ones when $k = 3$. Therefore, it is eventually blocked by the search engine. Cyclosa follows a more practical approach by spreading the load among the nodes with up to $94\,\mathrm{req/h}$ per node when $k = 3$, as shown in the simulation-based results presented in Figure 5.14, where X-Search's curve is split by the blocking threshold after only $5\,\mathrm{s}$ of the testing set execution.

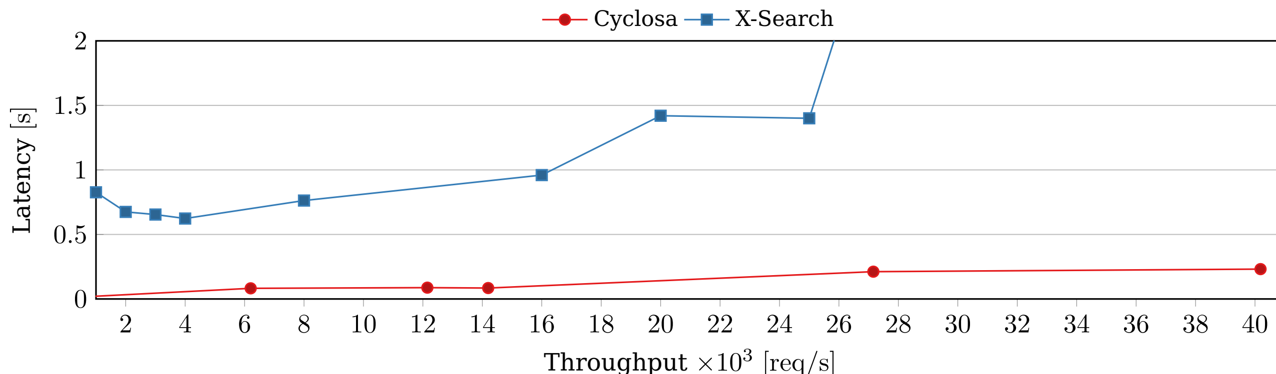

Figure 5.13: Throughput vs. latency of Cyclosa and X-Search.

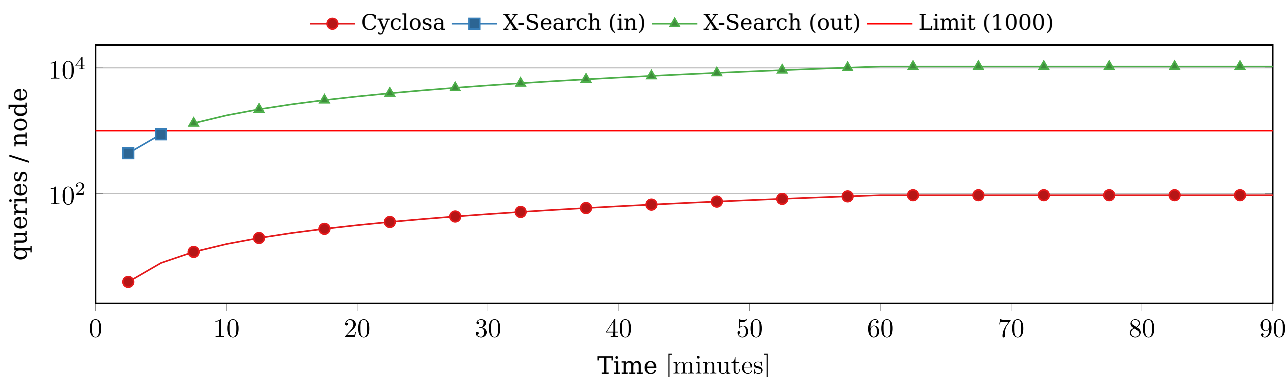

Figure 5.14: Query protection vs. users blocked by search engine.

Our results show that Cyclosa efficiently detects sensitive queries while limiting the number of overprotected requests which are not sensitive. We also show that it provides slightly better privacy protection than state-of-the-art solutions and drastically reduces the end-to-end latency without impact on accuracy. Particularly, Cyclosa:

(i) resists re-identification attacks better than alternatives with a low re-identification rate of $4\,\%$;

(ii) enables sub-second response times, which is on average $13\times$ faster than using Tor;

(iii) can sustain throughputs higher than $40.000\,\mathrm{req/s}$, enabling parallel users to securely browse the search engine; and

(iv) fairly balances the load between the participating nodes enabling all users to securely query the search engine without reaching its rate limitation.

5.3 Summary

Major service providers track user behaviour, which is a serious privacy threat in today's Internet. As one of the most widely used online services, search engines handle queries that may reveal sensitive information about individual users, such as sexual orientation, religion or political preferences. Existing solutions for enabling users to access Web search engines in a privacy-preserving manner do not tolerate strong adversaries or have poor performance. In this chapter, we explored the usage of TEEs to improve the capabilities of current solutions both in terms of privacy assurances and performance.

First, we proposed a novel architecture for privacy-preserving Web search which relies on SGX to support stronger adversarial models. X-Search (Section 5.1.1) operates as a proxy which stores and leverages user

past queries within a protected SGX enclave and generates obfuscated queries on the user's behalf. It does so by aggregating random past queries in such a way that the search engine is not able to distinguish which one is the original query, yet providing relevant results back to the user. Upon receiving a response from the search engine, X-Search filters results so that only those related to the initial query are forwarded.

We implemented a working prototype and evaluated it both analytically and experimentally using real-world datasets. Our observations indicated that X-Search can indeed provide accurate results without disclosing personal information about individuals. Most importantly, X-Search did so with a throughput improvement that reached orders of magnitude when compared to its predecessors, *i.e.*, PEAS and Tor.

The centralised nature of X-Search imposes some constraints though. The most notable being the easiness with which search engines could block requests originating from X-Search proxies as part of anti-bot mechanisms. To tackle that, we proposed Cyclosa (Section 5.2.1), a decentralised, private and accurate Web search solution that protects users against the risk of re-identification. It provides adaptive protection to different query sensitivity levels by combining the analysis of linkability to its issuer's profile and query semantic.

Cyclosa follows a fully decentralised architecture for higher scalability, and is based on SGX for preventing data leakage by relay nodes in its distributed setup. At the same time, Cyclosa reaches perfect accuracy by leveraging distinct paths for real and fake queries employed as obfuscation. In contrast to X-Search, it therefore obviates the need for finer grained filtering of results returned by search engines. Our implementation, evaluation and comparison to alternatives showed that Cyclosa is the most robust system against user re-identification and provided accurate responses to Web searches in an efficient and scalable way.

Chapter 6

Conclusion

Designing secure distributed systems is complex. Their dispersed nature multiplies the number of attack vectors. Apart from that, with the growth of cloud computing such systems tend to be at least partially deployed in shared infrastructures. This brings additional risks that stem from providers, their personnel or co-located tenants. Existing countermeasures are either infeasible due to computational complexity or fail at protecting running code and data from powerful adversaries like the operating system (OS). Because of that, the design of secure systems can greatly benefit from trusted execution environments (TEEs).

In this thesis, we have proposed several distributed architectures that leverage TEEs. We started in **Chapter 3** with systems that are likely to be deployed in cloud environments. For each of them, we isolate the minimum processing unit that handles sensitive data and put that component inside the trusted environment. In Section 3.1.1 that is done with the matcher component of a content-based routing (CBR) middleware. In this way, sensitive information within subscriptions and publications is safely matched inside enclaves before the publication payload is forwarded towards matching subscribers. We compare this approach with a software-only scheme with equivalent guarantees and reach better performances by one order of magnitude.

As for the lightweight MapReduce (Section 3.2.2) and SecureStreams (Section 3.2.3), we isolate the Lua scripting language's interpreter within enclaves. In such manner, encrypted code and data may enter the trusted environment, be deciphered and computed upon in plaintext form. Encrypted results are provided back to be handled by the underlying untrusted framework until reaching their final destination. This framework is what discerns each of the two systems. MapReduce runs atop of secure content-based routing (SCBR) for code and data dissemination and has distinct behaviour based on each phase of the MapReduce execution flow. SecureStreams, in turn, has a second Lua virtual machine (VM) running in untrusted mode for the execution of a distributed reactive programming pipeline. With all these systems we were able to observe performance losses due to software guard extensions (SGX), which are mostly influenced by memory usage.

In **Chapter 4**, we adopted a different design strategy. Instead of selecting subcomponents of frameworks to be placed inside secure enclaves, we take advantage of the isolation property they provide to design more efficient cryptographic protocols for group data sharing. The core mechanism relies on the generation of a master key in the secure environment where it is safely kept. Based on this premise, we can make use of simple and efficient cryptographic constructs, and occasionally simplify computationally complex phases in legacy protocols.

That is what we did both in IBBE-SGX (Section 4.1.1) and A-Sky (Section 4.2.1). IBBE-SGX simplifies the encryption step of an identity-based broadcast encryption (IBBE) scheme. Using the IBBE public key for such operation has quadratic complexity in the number of users in a group, while an equivalent formula that uses components of the master key makes the same with linear complexity. Since the master key is securely kept in the enclave, we are able to use the latter approach. This suffices to make the IBBE practical for demanding applications and to reap benefits such as considerably smaller ciphertexts and no need for public key infrastructure (PKI). We then tackled anonymity in group settings with A-Sky. To guarantee the anonymity, A-Sky uses a similar approach to GNU privacy guard (GPG) but using symmetric cryptography instead of asymmetric cryptography. This is possible because keys are maintained

by the trusted environment. Like IBBE-SGX, the advantages come both in performance improvements and shorter ciphertexts.

Chapter 5 brought the attention to users and showed how systems that rely on TEEs can protect their privacy against established service providers. TEEs help shielding user queries that cross by relays before reaching the provider's premises. Our first proposal, X-Search (Section 5.1.1), act as an intermediary between users and the Web search engine by proxying their requests, so that the provider cannot directly tell the user identity derived from the requester's Internet protocol (IP) address (unlinkability). Besides, it stores past user queries to obfuscate the requests by sending compound requests, so that the provider cannot tell the difference between a real query and a fake one (indistinguishability). With this approach, we offer better resistance against re-identification attacks while achieving better performance than alternatives.

Being a centralised system, X-Search does not scale. Because of that, we propose Cyclosa (Section 5.2.1), a peer-to-peer (P2P) network of users who are Web search clients and relays at the same time. These users, who have SGX-capable machines, issue queries intermediated by the Cyclosa enclave, which selects random peers to forward real and fake requests through distinct paths. Thanks to this, Cyclosa shows perfect accuracy of results as it just discards the ones coming from fake queries. Moreover, it adapts their number based on the real request's sensitivity, thus not overprotecting non-sensitive queries and generating less traffic. All combined, our evaluation shows that Cyclosa—the last of our contributions—achieves good performance while being scalable and delivering accurate results.

Looking back at our objectives enlisted in Section 1.1 in conjunction with the contributions of this work, we are able to draw some conclusions.

- How can TEEs help to achieve security and privacy in distributed systems?

  TEEs provide hardware isolation to applications. SGX does that by means of memory confidentiality and integrity guarantees of a running process' partition. The subcomponents of a system that handle sensitive data are good candidates to be part of this trusted partition that is able to crunch plaintext data.

  We show that for SCBR (matcher), SecureStreams and Lightweight MapReduce (Lua interpreter), IBBE-SGX and A-Sky (cryptographic operations), X-Search and Cyclosa (storage or forwarding of queries). The respective sensitive data consisted of publications and subscriptions (SCBR), code and processing data (SecureStreams and Lightweight MapReduce), keys (IBBE-SGX and A-Sky), and user queries (X-Search and Cyclosa). Whenever these data leave the enclave, they must be encrypted to be handled by untrusted code.

- What are the drawbacks?

  Since the OS is not trusted in the SGX threat model, enclaves cannot perform system calls. Because of that, running legacy applications in SGX enclaves might require considerable effort, which tends to be proportional to the amount of OS services those applications use. Although some runtime frameworks aiming to help with that were proposed (Section 2.2.6), we decided to design our systems from scratch in order to have control over the partitioning between trusted/untrusted code and to keep a minimised trusted computing base (TCB).

  Because of the need to track the integrity of memory pages, SGX has memory limitations (Section 2.2.1). There are two thresholds that once surpassed bring along performance overheads: the last level cache (LLC) and the enclave page cache (EPC) limits. We saw these drawbacks in SCBR both for oversubscribing the processor's cache (Figure 3.5) and the SGX protected memory (Figure 3.6) . In the Lightweight MapReduce we observed an increase in cache misses as the size of input data grows (Figure 3.14). In the other systems, we measured overall overheads of SGX in terms of latency and throughput for each solution.

- What benefits may come from using TEEs?

  The ability to securely process sensitive data in plaintext is key to achieve performance gains by replacing complex cryptographic constructs. We showed this for SCBR versus asymmetric scalar-product preserving encryption (ASPE), IBBE-SGX versus IBBE and A-Sky versus pretty good privacy

(PGP). All of them achieved better performances by at least one order of magnitude in comparison to the equivalent cryptographic scheme with no TEE. Additionally, both IBBE-SGX and A-Sky were able to produce a lot less metadata, entailing even further benefits in storage and network usage.

To possibly reduce infrastructure costs, the simple isolation of code and data enables the offloading of processing to untrusted environments, as we showed with the Lightweight MapReduce and SecureStreams. With Cyclosa and X-Search such isolation was used to guard users' sensitive data in order to protect their privacy. That was possible because the TEE assurances allow users to trust relays in order to obfuscate and convey their queries, *i.e.*, the X-Search proxy or the Cyclosa's client-side P2P network.

6.1 The road ahead

There are a number of interesting avenues for future work. Specific to our contributions, there are a few improvements that could be done. As we built a set of academic prototypes, many enhancements are required before turning them into production systems. We briefly comment on potential future work for each of our proposals.

Although subscriptions and publications are encrypted in SCBR, they are not integrity protected and could be replayed without detection. This could be solved by the addition of message authentication codes (MACs) and sequential numbers to each message. Additionally, in order to reduce enclave transitions, SCBR could benefit from processing publication batches instead of individual ones. The source code of SCBR is publicly available [11].

The Lua interpreter that runs inside the enclave in the Lightweight MapReduce and SecureStreams could be improved to provide additional security guarantees. It could, for instance, check the scripts for memory violations and other malicious behaviour. Just like SCBR, messages are not integrity- and replay-protected. The source code of both Lightweight MapReduce and SecureStreams is publicly available [13, 15].

IBBE-SGX could benefit from dynamically adapting the partition sizes based on the undergoing workload in order to balance speed optimisation of both administrator- and user-performed operations. For availability and fault tolerance, both IBBE-SGX and A-Sky would benefit from a distributed set of administrators. Some sort of distributed locking would have to be employed. Additionally, the possibility of auditing membership operations could be put in place with secure logs, possibly implemented with blockchains. The source code of A-Sky is publicly available [19].

X-Search and Cyclosa were implemented to work with a single search engine: Bing and Google, respectively. A more generic solution could be built by supporting more providers. Cyclosa's peer-to-peer network was implemented with a library that only supports local area networks. Obviously, real deployments should instead support world-wide deployments. Besides, both X-Search and Cyclosa would benefit from different browser integrations.

More generally, some security aspects could be further considered. In general, our prototypes did not implement attestation and key provisioning. When shared keys were not generated in enclaves, we assumed they were correctly provisioned, *i.e.*, after attestation (see Section 2.2.3). Key management systems are subject of active research, including involving SGX [253, 254].

Guaranteeing that a piece of persistent data is the most recent one is not an easy task, since stale data that is correctly encrypted and signed may be replayed by attackers. The enclaves in SCBR, A-Sky and X-Search persistently store active subscriptions, keys & membership information and user queries, respectively. They all would benefit from effective rollback prevention. Intel currently provides persistent monotonic counters for that, but they are known to be slow and vulnerable (see Section 2.2.5).

All in all, we believe that TEE came to stay and further research will certainly help to advance this exciting area. In this respect, the first commercially available machines with support for SGX2 were recently released. They have larger EPC, allow dynamic loading of memory pages and permit executions in release mode without keys provisioned by Intel. With regard to attacks, side channels will continue to be a hot topic for a while. We also believe that more applications that use TEEs will keep appearing, with special highlight to the domain of blockchains [255] and secure computing maketplaces [256].

Appendices

Appendix A

Publications

2016

1. Rafael Pires, Marcelo Pasin, Pascal Felber, and Christof Fetzer.
   Secure content-based routing using intel software guard extensions
   in Proceedings of the 17th International Middleware Conference, Middleware ’16.
   Trento, Italy: ACM, December 2016, pp. 10:1–10:10.
   doi: 10.1145/2988336.2988346.

2017

2. Florian Kelbert, Franz Gregor, Rafael Pires, Stefan Köpsell, Marcelo Pasin, Aurélien Havet, Valerio Schiavoni, Pascal Felber, Christof Fetzer, and Peter Pietzuch.
   SecureCloud: Secure big data processing in untrusted clouds
   in Design, Automation Test in Europe Conference Exhibition (DATE), 2017.
   Lausanne, Switzerland, March 2017, pp. 282–285.
   doi: 10.23919/DATE.2017.7926999.

3. Rafael Pires, Daniel Gavril, Pascal Felber, Emanuel Onica, and Marcelo Pasin.
   A lightweight MapReduce framework for secure processing with SGX
   in 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID).
   International Workshop on Assured Cloud Computing and QoS aware Big Data (WACC’17).
   Madrid, Spain, May 2017, pp. 1100–1107.
   doi: 10.1109/CCGRID.2017.129.

4. Aurélien Havet, Rafael Pires, Pascal Felber, Marcelo Pasin, Romain Rouvoy, and Valerio Schiavoni
   SecureStreams: A reactive middleware framework for secure data stream processing
   in Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS ’17.
   Barcelona, Spain: ACM, June 2017, pp. 124–133.
   doi: 10.1145/3093742.3093927.

5. Sonia Ben Mokhtar, Antoine Boutet, Pascal Felber, Marcelo Pasin, Rafael Pires, and Valerio Schiavoni
   X-search: Revisiting private web search using intel SGX
   in Proceedings of the 18th Middleware Conference, Middleware ’17.
   Las Vegas, USA: ACM, December 2017, pp. 198–208.
   doi: 10.1145/3135974.3135987.

2018

6. Stefan Contiu, Rafael Pires, Sébastien Vaucher, Marcelo Pasin, Pascal Felber, and Laurent Réveillère.
IBBE-SGX: Cryptographic group access control using trusted execution environments
in 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).
Luxembourg, June 2018, pp. 207–218.
doi: 10.1109/DSN.2018.00032.

7. Sébastien Vaucher, Rafael Pires, Pascal Felber, Marcelo Pasin, Valerio Schiavoni, and Christof Fetzer.
SGX-aware container orchestration for heterogeneous clusters
in 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS).
Vienna, Austria, July 2018, pp. 730– 741.
doi: 10.1109/ICDCS.2018.00076.

8. Rafael Pires, David Goltzsche, Sonia Ben Mokhtar, Sara Bouchenak, Antoine Boutet, Pascal Felber, Rüdiger Kapitza, Marcelo Pasin, and Valerio Schiavoni.
Cyclosa: Decentralizing private web search through SGX-based browser extensions
in 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS).
Vienna, Austria, July 2018, pp. 467–477.
doi: 10.1109/ICDCS.2018.00053.

9. Christian Göttel, Rafael Pires, Isabelly Rocha, Sébastien Vaucher, Pascal Felber, Marcelo Pasin, and Valerio Schiavoni.
Security, performance and energy trade-off of hardware-assisted memory protection mechanisms
in 2018 IEEE 37th International Symposium on Reliable Distributed Systems (SRDS).
Salvador, Brazil, October 2018, pp. 133-142.
doi: 10.1109/SRDS.2018.00024.

2019

10. Andrei Mogage, Rafael Pires, Crăciun Vlad, Emanuel Onica, and Pascal Felber.
Supply chain malware targets SGX: Take care of what you sign
in 2019 IEEE 38th International Symposium on Reliable Distributed Systems (SRDS).
Lyon, France, October 2019, pp. 52-60.
doi: 10.1109/SRDS.2019.00016

11. Stefan Contiu, Sébastien Vaucher, Rafael Pires, Marcelo Pasin, Pascal Felber, and Laurent Réveillère.
Anonymous and confidential file sharing over untrusted clouds
in 2019 IEEE 38th International Symposium on Reliable Distributed Systems (SRDS).
Lyon, France, October 2019, pp. 21-31.
doi: 10.1109/SRDS.2019.00013

Bibliography

[1] MITRE Corporation, *Common vulnerabilities and exposures (CVE) details: The ultimate security vulnerability datasource. browse vulnerabilities by date*, 2019. Accessed on 10/10/2019. [Online]. Available: https://www.cvedetails.com/browse-by-date.php.

[2] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures", *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974. doi: 10.1145/361011.361073.

[3] D. Snyder, *Intel unleashes next-gen enthusiast desktop pc platform at gamescom*, 2015. Accessed on 29/10/19. [Online]. Available: https://blogs.intel.com/technology/2015/08/6th-gen-gamescom/.

[4] A. Avizienis, J. Laprie, B. Randell and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing", *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004. doi: 10.1109/TDSC.2004.2.

[5] C. Göttel, R. Pires, I. Rocha, S. Vaucher, P. Felber, M. Pasin and V. Schiavoni, "Security, performance and energy trade-offs of hardware-assisted memory protection mechanisms", in *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS '18)*, Salvador, Brazil, 2018, pp. 133–142. doi: 10.1109/SRDS.2018.00024.

[6] A. Mogage, R. Pires, V. Crăciun, E. Onica and P. Felber, "Supply chain malware targets SGX: Take care of what you sign", in *2019 IEEE 38th Symposium on Reliable Distributed Systems (SRDS '19)*, Lyon, France, 2019. doi: 10.1109/SRDS.2019.00016.

[7] F. Kelbert, F. Gregor, R. Pires, S. Köpsell, M. Pasin, A. Havet, V. Schiavoni, P. Felber, C. Fetzer and P. Pietzuch, "SecureCloud: Secure big data processing in untrusted clouds", in *Design, Automation Test in Europe Conference Exhibition (DATE '17), 2017*, Lausanne, Switzerland, 2017, pp. 282–285. doi: 10.23919/DATE.2017.7926999.

[8] S. Vaucher, R. Pires, P. Felber, M. Pasin, V. Schiavoni and C. Fetzer, "SGX-aware container orchestration for heterogeneous clusters", in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS '18)*, Vienna, Austria, 2018, pp. 730–741. doi: 10.1109/ICDCS.2018.00076.

[9] S. Vaucher and R. Pires, *SGX-aware container orchestrator*, 2017. [Online]. Available: https://github.com/sebva/sgx-orchestrator.

[10] R. Pires, M. Pasin, P. Felber and C. Fetzer, "Secure content-based routing using intel software guard extensions", in *Proceedings of the 17th International Middleware Conference (Middleware '16)*, Trento, Italy: ACM, 2016, 10:1–10:10. doi: 10.1145/2988336.2988346.

[11] R. Pires, *Secure content-based routing (SCBR) using SGX enclaves*, 2016. [Online]. Available: https://github.com/rafaelppires/scbr.

[12] R. Pires, D. Gavril, P. Felber, E. Onica and M. Pasin, "A lightweight MapReduce framework for secure processing with SGX", in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID '17)*, Madrid, Spain, 2017, pp. 1100–1107. doi: 10.1109/CCGRID.2017.129.

[13] R. Pires, *Lightweight MapReduce: Lua interpreter inside SGX enclaves*, 2017. [Online]. Available: https://github.com/rafaelppires/sgx_lightweight_mapreruce.

[14] A. Havet, R. Pires, P. Felber, M. Pasin, R. Rouvoy and V. Schiavoni, "SecureStreams: A reactive middleware framework for secure data stream processing", in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems (DEBS '17)*, Barcelona, Spain: ACM, 2017, pp. 124–133. doi: 10.1145/3093742.3093927.

[15] A. Havet, R. Pires, and V. Schiavoni, *Securestreams, DEBS'17*, 2017. [Online]. Available: https://github.com/vschiavoni/SecureStreams-DEBS17.

[16] S. Contiu, R. Pires, S. Vaucher, M. Pasin, P. Felber and L. Réveillère, "IBBE-SGX: Cryptographic group access control using trusted execution environments", in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '18)*, Luxembourg city, Luxembourg, 2018, pp. 207–218. doi: 10.1109/DSN.2018.00032.

[17] R. Pires, and S. Contiu, *Trusted group file sharing over inquisitive cloud storages*, 2017. [Online]. Available: https://github.com/rafaelppires/trusted-sharing.

[18] S. Contiu, S. Vaucher, R. Pires, M. Pasin, P. Felber and L. Réveillère, “Anonymous and confidential file sharing over untrusted clouds”, in *2019 IEEE 38th Symposium on Reliable Distributed Systems (SRDS ’19)*, Lyon, France, 2019. doi: 10.1109/SRDS.2019.00013.

[19] R. Pires and S. Vaucher, *Anonymous and confidential file sharing over untrusted clouds*, 2018. [Online]. Available: https://github.com/rafaelppires/anonym-sharing.

[20] S. B. Mokhtar, A. Boutet, P. Felber, M. Pasin, R. Pires and V. Schiavoni, “X-search: Revisiting private web search using intel SGX”, in *Proceedings of the 18th ACM/IFIP/ USENIX Middleware Conference (Middleware ’17)*, Las Vegas, USA: ACM, 2017, pp. 198–208. doi: 10.1145/3135974.3135987.

[21] R. Pires, D. Goltzsche, S. Ben Mokhtar, S. Bouchenak, A. Boutet, P. Felber, R. Kapitza, M. Pasin and V. Schiavoni, “Cyclosa: Decentralizing private web search through SGX-based browser extensions”, in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS ’18)*, Vienna, Austria, 2018, pp. 467–477. doi: 10.1109/ICDCS.2018.00053.

[22] Intel, *Intel virtualization technology (intel VT)*, 2019. Accessed on 01/11/2019. [Online]. Available: https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html.

[23] AMD, *Amd-v technology for client virtualization*, 2019. Accessed on 01/11/2019. [Online]. Available: https://www.amd.com/en/technologies/virtualization.

[24] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch and S. Winwood, “seL4: Formal verification of an OS kernel”, in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP ’09)*, Big Sky, USA: ACM, 2009, pp. 207–220. doi: 10.1145/1629575.1629596.

[25] J. E. Gaffney, “Estimating the number of faults in code”, *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 459–464, 1984. doi: 10.1109/TSE.1984.5010260.

[26] K. Zetter, “NSA hacker chief explains how to keep him out of your system”, *Wired*, 2016. [Online]. Available: http://www.wired.com/2016/01/nsa-hacker-chief-explains-how-to-keep-him-out-of-your-system/.

[27] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum and E. W. Felten, “Lest we remember: Cold-boot attacks on encryption keys”, *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009. doi: 10.1145/1506409.1506429.

[28] M. Gruhn and T. Müller, “On the practicability of cold boot attacks”, in *2013 International Conference on Availability, Reliability and Security*, 2013, pp. 390–397. doi: 10.1109/ARES.2013.52.

[29] T. Dierks, *The Transport Layer Security (TLS) Protocol Version 1.2*, RFC 5246, 2015. doi: 10.17487/rfc5246. [Online]. Available: https://rfc-editor.org/rfc/rfc5246.txt.

[30] V. Costan, I. Lebedev and S. Devadas, “Sanctum: Minimal hardware extensions for strong software isolation”, in *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX, USA: USENIX Association, 2016, pp. 857–874. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan.

[31] D. Champagne and R. B. Lee, “Scalable architectural support for trusted software”, in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12. doi: 10.1109/HPCA.2010.5416657.

[32] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk and S. Devadas, “AEGIS: Architecture for tamper-evident and tamper-resistant processing”, in *Proceedings of the 17th Annual International Conference on Supercomputing*, ser. ICS ’03, San Francisco, USA: ACM, 2003, pp. 160–171. doi: 10.1145/782814.782838.

[33] M. Sabt, M. Achemlal and A. Bouabdallah, “The dual-execution-environment approach: Analysis and comparative evaluation”, in *ICT Systems Security and Privacy Protection*, H. Federrath and D. Gollmann, Eds., Cham: Springer International Publishing, 2015, pp. 557–570. doi: 10.1007/978-3-319-18467-8_37.

[34] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai and O. Mutlu, “Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors”, *SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014. doi: 10.1145/2678373.2665726.

[35] TCG: Trusted computing group, *Trusted platform modules (TPM) part 1, design principles*, 2011. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/TPM-Main-Part-1-Design-Principles_v1.2_rev116_01032011.pdf.

[36] *ARM security technology - building a secure system using TrustZone technology*, ARM Limited, 2009. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[37] S. Pinto and N. Santos, "Demystifying arm TrustZone: A comprehensive survey", *ACM Computing Surveys*, vol. 51, no. 6, 130:1–130:36, 2019. doi: 10.1145/3291047.

[38] Linaro, *Hardware unique key (HUK)*, Accessed on 17/09/2019. [Online]. Available: https://optee.readthedocs.io/en/latest/architecture/porting_guidelines.html.

[39] N. Santos, H. Raj, S. Saroiu and A. Wolman, "Using ARM trustzone to build a trusted language runtime for mobile applications", in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14, Salt Lake City, USA: ACM, 2014, pp. 67–80. doi: 10.1145/2541940.2541949.

[40] S. Zhao, Q. Zhang, G. Hu, Y. Qin and D. Feng, "Providing root of trust for arm trustzone using on-chip sram", in *Proceedings of the 4th International Workshop on Trustworthy Embedded Devices*, ser. TrustED '14, Scottsdale, USA: ACM, 2014, pp. 25–36. doi: 10.1145/2666141.2666145.

[41] S. Gueron, "A memory encryption engine suitable for general purpose processors", *IACR Cryptology ePrint Archive, Report 2016/204*, 2016. [Online]. Available: https://eprint.iacr.org/2016/204.pdf.

[42] Intel, *Intel software guard extensions SDK developer reference for Linux OS*, version 2.5, 2019. [Online]. Available: https://download.01.org/intel-sgx/linux-2.5/docs/Intel_SGX_Developer_Reference_Linux_2.5_Open_Source.pdf.

[43] H. Tian, Q. Zhang, S. Yan, A. Rudnitsky, L. Shacham, R. Yariv and N. Milshten, "Switchless calls made practical in intel SGX", in *Proceedings of the 3rd Workshop on System Software for Trusted Execution (SysTEX '18)*, Toronto, Canada, 2018, pp. 22–27. doi: 10.1145/3268935.3268942.

[44] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd and C. Rozas, "Intel software guard extensions (intel SGX) support for dynamic memory management inside an enclave", in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016 (HASP '16)*, Seoul, Republic of Korea: ACM, 2016, 10:1–10:9. doi: 10.1145/2948618.2954331.

[45] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch and R. Kapitza, "SecureKeeper: Confidential ZooKeeper using intel SGX", in *Proceedings of the 17th International Middleware Conference (Middleware '16)*, Trento, Italy: ACM, 2016, 14:1–14:13. doi: 10.1145/2988336.2988350.

[46] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch and C. Fetzer, "SCONE: Secure linux containers with intel SGX", in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, USA: USENIX Association, 2016, pp. 689–703. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.

[47] R. Elbaz, D. Champagne, C. Gebotys, R. B. Lee, N. Potlapally and L. Torres, "Hardware mechanisms for memory authentication: A survey of existing techniques and engines", *Transactions on Computational Science IV, Lecture Notes in Computer Science (LNCS)*, pp. 1–22, 2009. doi: 10.1007/978-3-642-01004-0_1.

[48] L. Soares and M. Stumm, "FlexSC: Flexible system call scheduling with exception-less system calls", in *9th Usenix Conference on Operating Systems Design and Implementation (OSDI'10)*, Vancouver, BC, Canada, 2010, pp. 33–46. [Online]. Available: https://www.usenix.org/conference/osdi10/flexsc-flexible-system-call-scheduling-exception-less-system-calls.

[49] O. Weisse, V. Bertacco and T. Austin, "Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves", in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*, Toronto, Canada: ACM, 2017, pp. 81–93. doi: 10.1145/3079856.3080208. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080208.

[50] M. Orenbach, P. Lifshits, M. Minkin and M. Silberstein, "Eleos: Exitless OS services for SGX enclaves", in *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*, Belgrade, Serbia: ACM, 2017, pp. 238–253. doi: 10.1145/3064176.3064219.

[51] I. Anati, S. Gueron, S. Johnson and V. Scarlata, "Innovative technology for CPU based attestation and sealing", in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '13)*, Tel-Aviv, Israel, 2013. [Online]. Available: https://software.intel.com/sites/default/files/article/413939/hasp-2013-innovative-technology-for-attestation-and-sealing.pdf.

[52] V. Costan and S. Devadas, *Intel SGX explained*, Cryptology ePrint Archive, Report 2016/086, 2016. [Online]. Available: https://eprint.iacr.org/2016/086.

[53] Positive Technologies, *Disabling intel ME 11 via undocumented mode*, 2017. Accessed on 07/10/2019. [Online]. Available: http://blog.ptsecurity.com/2017/08/disabling-intel-me.html.

[54] R. Strackx and F. Piessens, "Ariadne: A minimal approach to state continuity", in *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX: USENIX Association, 2016, pp. 875–892. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/strackx.

[55] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens and J. M. McCune, “Memoir: Practical state continuity for protected modules”, in *2011 IEEE Symposium on Security and Privacy*, 2011, pp. 379–394. doi: 10.1109/SP.2011.38.

[56] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels and S. Capkun, “ROTE: Rollback protection for trusted execution”, in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, Canada: USENIX Association, 2017, pp. 1289–1306. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/matetic.

[57] M. Bailleu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda and K. Vaswani, “SPEICHER: Securing LSM-based key-value stores using shielded execution”, in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, Boston, MA, 2019, pp. 173–190. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/bailleu.

[58] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber and C. Fetzer, “SGXBOUNDS: Memory safety for shielded execution”, in *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys ’17)*, Belgrade, Serbia, 2017, pp. 205–221. doi: 10.1145/3064176.3064192.

[59] N. Weichbrodt, A. Kurmus, P. Pietzuch and R. Kapitza, “Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves”, in *Computer Security – ESORICS 2016*, Cham: Springer International Publishing, 2016, pp. 440–457. doi: 10.1007/978-3-319-45744-4_22.

[60] H. Shacham, “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)”, in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS ’07, Alexandria, Virginia, USA, 2007, pp. 552–561. doi: 10.1145/1315245.1315313.

[61] T. Bletsch, X. Jiang, V. W. Freeh and Z. Liang, “Jump-oriented programming: A new class of code-reuse attack”, in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS ’11, Hong Kong, China: ACM, 2011, pp. 30–40. doi: 10.1145/1966913.1966919. [Online]. Available: http://doi.acm.org/10.1145/1966913.1966919.

[62] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado and B. B. Kang, “Hacking in darkness: Return-oriented programming against secure enclaves”, in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC, 2017, pp. 523–539. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-jaehyuk.

[63] “SGX-shield: Enabling address space layout randomization for SGX programs”, in *Network and Distributed System Security Symposium (NDSS ’17)*, 2017. doi: 10.14722/ndss.2017.23037.

[64] A. Biondo, M. Conti, L. Davi, T. Frassetto and A.-R. Sadeghi, “The guard’s dilemma: Efficient code-reuse attacks against intel SGX”, in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, 2018, pp. 1213–1227. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/biondo.

[65] A. Baumann, M. Peinado and G. Hunt, “Shielding applications from an untrusted cloud with haven”, in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO: USENIX Association, 2014, pp. 267–283. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann.

[66] C. che Tsai, D. E. Porter and M. Vij, “Graphene-SGX: A practical library OS for unmodified applications on SGX”, in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA: USENIX Association, 2017, pp. 645–658. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

[67] S. Shinde, D. L. Tien, S. Tople and P. Saxena, “Panoply: Low-TCB linux applications with SGX enclaves”, in *Network and Distributed System Security Symposium (NDSS ’17)*, 2017. doi: 10.14722/ndss.2017.23500.

[68] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov and P. R. Pietzuch, “SGX-LKL: securing the host OS interface for trusted execution”, *CoRR*, vol. abs/1908.11143, 2019. [Online]. Available: http://arxiv.org/abs/1908.11143.

[69] Y. Xu, W. Cui and M. Peinado, “Controlled-channel attacks: Deterministic side channels for untrusted operating systems”, in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 640–656. doi: 10.1109/SP.2015.45.

[70] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun and A.-R. Sadeghi, “Software grand exposure: SGX cache attacks are practical”, in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC: USENIX Association, 2017. [Online]. Available: https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser.

[71] Intel, *Intel SGX and side-channels*, 2017. Accessed on 09/10/2019. [Online]. Available: https://software.intel.com/en-us/articles/intel-sgx-and-side-channels.

[72] M.-W. Shih, S. Lee, T. Kim and M. Peinado, “T-SGX: Eradicating controlled-channel attacks against enclave programs”, in *Network and Distributed System Security Symposium (NDSS ’17)*, 2017. doi: 10.14722/ndss.2017.23193.

[73] Y. Fu, E. Bauman, R. Quinonez and Z. Lin, “SGX-LAPD: Thwarting controlled side channel attacks via enclave verifiable page faults”, in *Research in Attacks, Intrusions, and Defenses*, Cham: Springer International Publishing, 2017, pp. 357–380. doi: 10.1007/978-3-319-66332-6_16.

[74] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz and Y. Yarom, “Spectre attacks: Exploiting speculative execution”, in *2019 IEEE Symposium on Security and Privacy (SP ’19)*, San Francisco, USA, 2019. [Online]. Available: https://arxiv.org/pdf/1801.01203.pdf.

[75] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin and T. H. Lai, “SgxPectre: Stealing intel secrets from SGX enclaves via speculative execution”, in *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, 2019, pp. 142–157. doi: 10.1109/EuroSP.2019.00020.

[76] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer and A. Kurmus, “SMoTherSpectre: Exploiting speculative execution through port contention”, in *2019 ACM SIGSAC Conference on Computer & Communications Security (CCS ’19)*, 2019. doi: 10.1145/3319535.3363194.

[77] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom and M. Hamburg, “Meltdown: Reading kernel memory from user space”, in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, USA, 2018, pp. 973–990. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[78] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom and R. Strackx, “Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution”, in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, USA: USENIX Association, 2018, 991–1008. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[79] A. Brito, C. Fetzer, S. Köpsell, M. Pasin, P. Felber, K. Fonseca, M. Rosa, L. Gomes, R. Riella, C. Prado, L. F. R. da Costa Carmo, D. Lucani, M. Sipos, L. Nagy and M. Fehér, “Cloud challenge: Secure end-to-end processing of smart metering data”, in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion ’18)*, Zurich, Switzerland, 2018, pp. 36–42. doi: 10.1109/UCC-Companion.2018.00031.

[80] J. Lind, C. Priebe, D. Muthukumaran, D. O’Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer and P. Pietzuch, “Glamdring: Automatic application partitioning for intel SGX”, in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, USA: USENIX Association, 2017, pp. 285–298. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind.

[81] P.-L. Aublin, F. Kelbert, D. O’Keeffe, D. Muthukumaran, C. Priebe, J. Lind, R. Krahn, C. Fetzer, D. Eyers and P. Pietzuch, “Libseal: Revealing service integrity violations using trusted execution”, in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys ’18, Porto, Portugal, 2018, 24:1–24:15. doi: 10.1145/3190508.3190547.

[82] SecureCloud consortium, *Like coding in the clouds*, 2019. Accessed on 05/11/2019. [Online]. Available: https://www.securecloudproject.eu/.

[83] P. T. Eugster, P. A. Felber, R. Guerraoui and A.-M. Kermarrec, “The many faces of publish/subscribe”, *ACM Computing Surveys*, vol. 35, no. 2, pp. 114–131, 2003. doi: 10.1145/857076.857078.

[84] A. Carzaniga, D. S. Rosenblum and A. L. Wolf, “Design and evaluation of a wide-area event notification service”, *ACM Transactions on Computer Systems (TOCS)*, vol. 19, no. 3, pp. 332–383, 2001. doi: 10.1145/380749.380767.

[85] R. Chand and P. Felber, “XNET: A reliable content-based publish/subscribe system”, in *23rd International Symposium on Reliable Distributed Systems (SRDS 2004)*, 2004, pp. 264–273. doi: 10.1109/RELDIS.2004.1353027.

[86] G. Li, S. Hou and H.-A. Jacobsen, “A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams”, in *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, ser. ICDCS ’05, Washington, DC, USA: IEEE Computer Society, 2005, pp. 447–457. doi: 10.1109/ICDCS.2005.8.

[87] E. Onica, P. Felber, H. Mercier and E. Rivière, “Confidentiality-preserving publish/ subscribe: A survey”, *ACM Computing Surveys (CSUR)*, vol. 49, no. 2, 27:1–27:43, 2016. doi: 10.1145/2940296.

[88] A. V. Uzunov, “A survey of security solutions for distributed publish/subscribe systems”, *Computers & Security*, vol. 61, no. C, pp. 94–129, 2016. doi: 10.1016/j.cose.2016.04.008.

[89] S. Choi, G. Ghinita and E. Bertino, "A privacy-enhancing content-based publish/subscribe system using scalar product preserving transformations", in *Proceedings of the 21st International Conference on Database and Expert Systems Applications: Part I (DEXA'10)*, Bilbao, Spain: Springer-Verlag, 2010, pp. 368–384. [Online]. Available: http://dl.acm.org/citation.cfm?id=1881867.1881908.

[90] R. Barazzutti, P. Felber, H. Mercier, E. Onica and E. Rivière, "Thrifty privacy: Efficient support for privacy-preserving publish/subscribe", in *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems (DEBS '12)*, Berlin, Germany: ACM, 2012, pp. 225–236. doi: 10.1145/2335484.2335509.

[91] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors", *Communications of the ACM*, vol. 13, pp. 422–426, 1970.

[92] J. Bacon, D. M. Eyers, J. Singh and P. R. Pietzuch, "Access control in publish/subscribe systems", in *Proceedings of the Second International Conference on Distributed Event-based Systems (DEBS '08)*, Rome, Italy: ACM, 2008, pp. 23–34. doi: 10.1145/1385989.1385993.

[93] Y. Zhao and D. C. Sturman, "Dynamic access control in a content-based publish/subscribe system with delivery guarantees", in *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, ser. ICDCS '06, Washington, DC, USA: IEEE Computer Society, 2006. doi: 10.1109/ICDCS.2006.32.

[94] A. Wun and H.-A. Jacobsen, "A policy management framework for content-based publish/subscribe middleware", in *Proceedings of Middleware 2007: ACM/IFIP/USENIX 8th International Middleware Conference*, Berlin, Germany: Springer Berlin Heidelberg, 2007, pp. 368–388. doi: 10.1007/978-3-540-76778-7_19.

[95] L. Sampaio, F. Silva, A. Souza, A. Brito and P. Felber, "Secure and privacy-aware data dissemination for cloud-based applications", in *10th IEEE/ACM International Conference on Utility and Cloud Computing (UCC'17)*, Austin, USA. doi: 10.1145/3147213.3147230.

[96] S. Arnautov, A. Brito, P. Felber, C. Fetzer, F. Gregor, R. Krahn, W. Ozga, A. Martin, V. Schiavoni, F. Silva, M. Tenorio and N. Thümmel, "PubSub-SGX: Exploiting trusted execution environments for privacy-preserving publish/subscribe systems", in *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS '18)*, 2018, pp. 123–132. doi: 10.1109/SRDS.2018.00023.

[97] W. Wei, J. Du, T. Yu and X. Gu, "SecureMR: A service integrity assurance framework for mapreduce", in *Proceedings of the 2009 Annual Computer Security Applications Conference*, ser. ACSAC '09, Washington, DC, USA: IEEE Computer Society, 2009, pp. 73–82. doi: 10.1109/ACSAC.2009.17.

[98] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov and E. Witchel, "Airavat: Security and privacy for MapReduce", in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'10, San Jose, California: USENIX Association, 2010, pp. 20–20. [Online]. Available: https://www.usenix.org/conference/nsdi10-0/airavat-security-and-privacy-mapreduce.

[99] P. Mohan, A. Thakurta, E. Shi, D. Song and D. Culler, "GUPT: Privacy Preserving Data Analysis Made Easy", in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*, Scottsdale, Arizona, USA: ACM, 2012, pp. 349–360. doi: 10.1145/2213836.2213876.

[100] C. Dwork, F. McSherry, K. Nissim and A. Smith, "Calibrating noise to sensitivity in private data analysis", in *Theory of Cryptography*, Berlin, Germany: Springer Berlin Heidelberg, 2006, pp. 265–284. doi: 10.1007/11681878_14.

[101] S. D. Tetali, M. Lesani, R. Majumdar and T. Millstein, "MrCrypt: Static analysis for secure cloud computations", in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages Applications (OOPSLA '13)*, Indianapolis, USA: ACM, 2013, pp. 271–286. doi: 10.1145/2509136.2509554.

[102] K. Zhang, X. Zhou, Y. Chen, X. Wang and Y. Ruan, "Sedic: Privacy-aware data intensive computing on hybrid clouds", in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*, Chicago, Illinois, USA: ACM, 2011, pp. 515–526. doi: 10.1145/2046707.2046767.

[103] C. Zhang, E. Chang and R. H. C. Yap, "Tagged-mapreduce: A general framework for secure computing with mixed-sensitivity data on hybrid clouds", in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014, pp. 31–40. doi: 10.1109/CCGrid.2014.96.

[104] X. Xu and X. Zhao, "A framework for privacy-aware computing on hybrid clouds with mixed-sensitivity data", in *2015 IEEE International Symposium on Big Data Security on Cloud*, 2015, pp. 1344–1349. doi: 10.1109/HPCC-CSS-ICESS.2015.110.

[105] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz and M. Russinovich, "VC3: Trustworthy data analytics in the cloud using SGX", in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 38–54. doi: 10.1109/SP.2015.10.

[106] O. Ohrimenko, M. Costa, C. Fournet, C. Gkantsidis, M. Kohlweiss and D. Sharma, “Observing and preventing leakage in MapReduce”, in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS ’15)*, Denver, Colorado, USA: ACM, 2015, pp. 1570–1581. doi: 10.1145/2810103.2813695.

[107] T. T. A. Dinh, P. Saxena, E.-C. Chang, B. C. Ooi and C. Zhang, “M2R: Enabling stronger privacy in MapReduce computation”, in *24th USENIX Security Symposium (USENIX Security 15)*, Washington, D.C.: USENIX Association, 2015, pp. 447–462. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/dinh.

[108] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy and S. Taneja, “Twitter Heron: Stream processing at scale”, in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’15, Melbourne, Australia: ACM, 2015, pp. 239–250. doi: 10.1145/2723372.2742788.

[109] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt and S. Whittle, “The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing”, *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1792–1803, 2015. doi: 10.14778/2824032.2824076.

[110] *Spark Streaming*, https://spark.apache.org/streaming, Accessed on 20/05/2019.

[111] *Apache Storm*, http://storm.apache.org, Accessed on 20/05/2019.

[112] *Infinispan*, http://infinispan.org, Accessed on 20/05/2019.

[113] T. Lindholm, F. Yellin, G. Bracha and A. Buckley, *The Java Virtual Machine Specification, Java SE 8 Edition*, 1st. Addison-Wesley Professional, 2014.

[114] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker and I. Stoica, “Discretized streams: Fault-tolerant streaming computation at scale”, in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP ’13, Farminton, Pennsylvania, USA: ACM, 2013, pp. 423–438. doi: 10.1145/2517349.2522737.

[115] S. Y. Shah, B. Paulovicks and P. Zerfos, “Data-at-rest security for spark”, in *2016 IEEE International Conference on Big Data (Big Data)*, 2016, pp. 1464–1473. doi: 10.1109/BigData.2016.7840754.

[116] L. L.-S. D. S. Group, *Sgx-spark*, London, UK, 2019. Accessed on 16/10/2019. [Online]. Available: https://github.com/lsds/sgx-spark.

[117] D. Le Quoc, F. Gregor, J. Singh and C. Fetzer, “SGX-PySpark: Secure distributed data analytics”, in *The World Wide Web Conference (WWW ’19)*, San Francisco, CA, USA, 2019, pp. 3564–3563. doi: 10.1145/3308558.3314129. [Online]. Available: http://doi.acm.org/10.1145/3308558.3314129.

[118] J. J. Stephen, S. Savvides, V. Sundaram, M. S. Ardekani and P. Eugster, “STYX: Stream Processing with Trustworthy Cloud-based Execution”, in *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC ’16)*, Santa Clara, CA, USA: ACM, 2016, pp. 348–360. doi: 10.1145/2987550.2987574.

[119] Enno and more, *Reactive streams for Kafka*, Accessed on 20/05/2019. [Online]. Available: https://github.com/akka/reactive-kafka.

[120] *Apache Kafka*, https://kafka.apache.org, Accessed on 20/05/2019.

[121] J. Kreps, N. Narkhede and J. Rao, “Kafka: A distributed messaging system for log processing”, in *Networking meets databases (NetDB ’11)*, Athens, Greece, 2011, pp. 1–7. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2017/09/Kafka.pdf.

[122] H. Park, S. Zhai, L. Lu and F. X. Lin, “Streambox-tz: Secure stream analytics at the edge with trustzone”, in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA: USENIX Association, 2019, pp. 537–554. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/park-heejin.

[123] W. C. Garrison, A. Shull, S. Myers and A. J. Lee, “On the practicality of cryptographically enforcing dynamic access control policies in the cloud”, in *2016 IEEE Symposium on Security and Privacy (SP)*, San Jose, USA, 2016, pp. 819–838. doi: 10.1109/SP.2016.54.

[124] A. Sahai and B. Waters, “Fuzzy identity-based encryption”, in *Advances in Cryptology – EUROCRYPT 2005*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 457–473. doi: 10.1007/11426639_27.

[125] V. Goyal, O. Pandey, A. Sahai and B. Waters, “Attribute-based encryption for fine-grained access control of encrypted data”, in *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS ’06)*, Alexandria, Virginia, USA: ACM, 2006, pp. 89–98. doi: 10.1145/1180405.1180418.

[126] J. Bethencourt, A. Sahai and B. Waters, “Ciphertext-policy attribute-based encryption”, in *2007 IEEE Symposium on Security and Privacy (SP ’07)*, 2007, pp. 321–334. doi: 10.1109/SP.2007.11.

[127] D. Boneh, X. Boyen and E.-J. Goh, "Hierarchical identity based encryption with constant size ciphertext", in *Advances in Cryptology – EUROCRYPT 2005*, Berlin, Germany: Springer, 2005, pp. 440–456. doi: 10.1007/11426639_26.

[128] D. Boneh, A. Sahai and B. Waters, "Functional encryption: Definitions and challenges", in *Theory of Cryptography*, Berlin, Germany: Springer, 2011, pp. 253–273. doi: 10.1007/978-3-642-19571-6_16.

[129] B. Fisch, D. Vinayagamurthy, D. Boneh and S. Gorbunov, "Iron: Functional encryption using intel sgx", in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, Dallas, Texas, USA: ACM, 2017, pp. 765–782. doi: 10.1145/3133956.3134106.

[130] G. Ateniese, K. Fu, M. Green and S. Hohenberger, "Improved proxy re-encryption schemes with applications to secure distributed storage", *ACM Transactions on Information and System Security*, vol. 9, no. 1, pp. 1–30, 2006. doi: 10.1145/1127345.1127346.

[131] M. Green and G. Ateniese, "Identity-based proxy re-encryption", in *Proceedings of the 5th International Conference on Applied Cryptography and Network Security (ACNS '07)*, Zhuhai, China: Springer-Verlag, 2007, pp. 288–306. doi: 10.1007/978-3-540-72738-5_19.

[132] M. Green, S. Hohenberger and B. Waters, "Outsourcing the decryption of ABE ciphertexts", in *Proceedings of the 20th USENIX Conference on Security (SEC'11)*, San Francisco, CA: USENIX Association, 2011, pp. 34–34. [Online]. Available: http://static.usenix.org/event/sec11/tech/full_papers/Green.pdf.

[133] A. Sahai, H. Seyalioglu and B. Waters, "Dynamic credentials and ciphertext delegation for attribute-based encryption", in *Advances in Cryptology – CRYPTO 2012*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 199–217. doi: 10.1007/978-3-642-32009-5_13.

[134] D. R. Stinson, *Cryptography: theory and practice*. CRC press, 2005.

[135] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor and B. Pinkas, "Multicast security: A taxonomy and some efficient constructions", in *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings.*, vol. 2, 1999, 708–716 vol.2. doi: 10.1109/INFCOM.1999.751457.

[136] D. Wallner, E. Harder and R. Agee, *Key management for multicast: Issues and architectures*, RFC 2627, United States, 1999. doi: 10.17487/rfc2627.

[137] A. Fiat and M. Naor, "Broadcast encryption", in *Advances in Cryptology — CRYPTO' 93*, Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 480–491. doi: 10.1007/3-540-48329-2_40.

[138] M. Naor and B. Pinkas, "Efficient trace and revoke schemes", *International Journal of Information Security*, vol. 9, no. 6, pp. 411–424, 2010. doi: 10.1007/s10207-010-0121-2.

[139] A. Bessani, M. Correia, B. Quaresma, F. André and P. Sousa, "Depsky: Dependable and secure storage in a cloud-of-clouds", *ACM Transactions on Storage (TOS)*, vol. 9, no. 4, 12:1–12:33, 2013. doi: 10.1145/2535929.

[140] A. Shamir, "How to share a secret", *Communication of the ACM*, vol. 22, no. 11, pp. 612–613, 1979. doi: 10.1145/359168.359176.

[141] A. Bessani, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin and P. Verissimo, "SCFS: A shared cloud-backed file system", in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA: USENIX Association, 2014, pp. 169–180. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/bessani.

[142] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang and L. Zhuang, "Enabling security in cloud storage SLAs with CloudProof", in *Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC 11)*, Portland, USA: USENIX Association, 2011, pp. 31–31. [Online]. Available: https://www.usenix.org/conference/usenixatc11/enabling-security-cloud-storage-slas-cloudproof.

[143] D. Boneh, C. Gentry and B. Waters, "Collusion resistant broadcast encryption with short ciphertexts and private keys", in *Advances in Cryptology – CRYPTO 2005*, Berlin, Germany: Springer, 2005, pp. 258–275. doi: 10.1007/11535218_16.

[144] F. Wang, J. Mickens, N. Zeldovich and V. Vaikuntanathan, "Sieve: Cryptographically enforced access control for user data in untrusted clouds", in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA: USENIX Association, 2016, pp. 611–626. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/wang-frank.

[145] J. Li, C. Qin, P. P. C. Lee and J. Li, "Rekeying for encrypted deduplication storage", in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016, pp. 618–629. doi: 10.1109/DSN.2016.62.

[146] P. Rösler, C. Mainka and J. Schwenk, "More is less: On the end-to-end security of group chats in Signal, WhatsApp, and Threema", in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018, pp. 415–429. doi: 10.1109/EuroSP.2018.00036.

[147] S. Angel and S. Setty, “Unobservable communication over fully untrusted infrastructure”, in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA: USENIX Association, 2016, pp. 551–569. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/angel.

[148] P. R. Zimmermann, *The Official PGP User’s Guide*. Cambridge, MA, USA: MIT Press, 1995.

[149] A. Barth, D. Boneh and B. Waters, “Privacy in encrypted content distribution using private broadcast encryption”, in *Financial Cryptography and Data Security*, Berlin, Germany: Springer Berlin Heidelberg, 2006, pp. 52–64. doi: 10.1007/11889663_4.

[150] D. Boneh, E. Shen and B. Waters, “Strongly unforgeable signatures based on computational diffie-hellman”, in *Public Key Cryptography - PKC 2006*, Berlin, Germany: Springer Berlin Heidelberg, 2006, pp. 229–240. doi: 10.1007/11745853_15.

[151] B. Libert, K. G. Paterson and E. A. Quaglia, “Anonymous broadcast encryption: Adaptive security and efficient constructions in the standard model”, in *Public Key Cryptography – PKC 2012*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 206–224. doi: 10.1007/978-3-642-30057-8_13.

[152] P. MacKenzie, M. K. Reiter and K. Yang, “Alternatives to non-malleability: Definitions, constructions, and applications”, in *Theory of Cryptography*, Berlin, Germany: Springer Berlin Heidelberg, 2004, pp. 171–190. doi: 10.1007/978-3-540-24638-1_10.

[153] C. Aguilar-Melchor, J. Barrier, L. Fousse and M.-O. Killijian, “XPIR: Private information retrieval for everyone”, *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 2, pp. 155 –174, 2016. doi: 10.1515/popets-2016-0010.

[154] R. Dingledine, N. Mathewson and P. F. Syverson, “Tor: The second-generation onion router”, in *Proceedings of the 13th USENIX Security Symposium*, San Diego, USA, 2004, pp. 303–320. [Online]. Available: http://www.usenix.org/publications/library/proceedings/sec04/tech/dingledine.html.

[155] D. Goldschlag, M. Reed and P. Syverson, “Onion routing”, *Communications of the ACM*, vol. 42, no. 2, pp. 39–41, 1999. doi: 10.1145/293411.293443.

[156] S. B. Mokhtar, G. Berthou, A. Diarra, V. Quéma and A. Shoker, “RAC: A freerider-resilient, scalable, anonymous communication protocol”, in *2013 IEEE 33rd International Conference on Distributed Computing Systems (ICDCS ’13)*, 2013, pp. 520–529. doi: 10.1109/ICDCS.2013.52.

[157] H. Corrigan-Gibbs and B. Ford, “Dissent: Accountable anonymous group messaging”, in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS ’10)*, Chicago, Illinois, USA: ACM, 2010, pp. 340–350. doi: 10.1145/1866307.1866346.

[158] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford and A. Johnson, “Dissent in numbers: Making strong anonymity scale”, in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, Hollywood, CA: USENIX, 2012, pp. 179–182. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/wolinsky.

[159] D. Chaum, “The dining cryptographers problem: Unconditional sender and recipient untraceability”, *Journal of Cryptology*, vol. 1, no. 1, pp. 65–75, 1988. doi: 10.1007/BF00206326. [Online]. Available: https://doi.org/10.1007/BF00206326.

[160] J. Brickell and V. Shmatikov, “Efficient anonymity-preserving data collection”, in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD ’06)*, Philadelphia, PA, USA: ACM, 2006, pp. 76–85. doi: 10.1145/1150402.1150415.

[161] S. T. Peddinti and N. Saxena, “Web search query privacy: Evaluating query obfuscation and anonymizing networks”, *Journal of Computer Security*, vol. 22, no. 1, pp. 155–199, 2014. doi: 10.3233/JCS-130491.

[162] A. Petit, T. Cerqueus, A. Boutet, S. B. Mokhtar, D. Coquil, L. Brunie and H. Kosch, “SimAttack: Private web search under fire”, *Journal of Internet Services and Applications*, vol. 7, no. 1, p. 2, 2016. doi: 10.1186/s13174-016-0044-x.

[163] A. Arampatzis, P. S. Efraimidis and G. Drosatos, “A query scrambler for search privacy on the internet”, *Information Retrieval*, vol. 16, no. 6, pp. 657–679, 2013. doi: 10.1007/s10791-012-9212-1.

[164] D. C. Howe and H. Nissenbaum, “TrackMeNot: Resisting surveillance in web search”, *Lessons from the Identity Trail: Anonymity, Privacy, and Identity in a Networked Society*, vol. 23, pp. 417–436, 2009.

[165] J. Domingo-Ferrer, A. Solanas and J. Castellà-Roca, “H(k)-private information retrieval from privacy-uncooperative queryable databases”, *Online Information Review*, vol. 33, no. 4, pp. 720–744, 2009. doi: 10.1108/14684520910985693.

[166] A. Petit, T. Cerqueus, S. B. Mokhtar, L. Brunie and H. Kosch, “PEAS: Private, efficient and accurate web search”, in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1, 2015, pp. 571–580. doi: 10.1109/Trustcom.2015.421.

[167] G. Pass, A. Chowdhury and C. Torgeson, "A Picture of Search", in *Proceedings of the 1st International Conference on Scalable Information Systems (InfoScale '06)*, Hong Kong: ACM, 2006. doi: 10.1145/1146847.1146848.

[168] D. Goltzsche, C. Wulf, D. Muthukumaran, K. Rieck, P. Pietzuch and R. Kapitza, "TrustJS: Trusted client-side execution of JavaScript", in *Proceedings of the 10th European Workshop on Systems Security (EuroSec'17)*, Belgrade, Serbia: ACM, 2017, 7:1–7:6. doi: 10.1145/3065913.3065917.

[169] S. Kim, Y. Shin, J. Ha, T. Kim and D. Han, "A first step towards leveraging commodity trusted execution environments for network applications", in *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets)*, Philadelphia, PA, USA: ACM, 2015, 7:1–7:7. doi: 10.1145/2834050.2834100.

[170] R. Barazzutti, P. Felber, C. Fetzer, E. Onica, J.-F. Pineau, M. Pasin, E. Rivière and S. Weigert, "StreamHub: A massively parallel architecture for high-performance content-based publish/subscribe", in *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems (DEBS '13)*, Arlington, Texas, USA: ACM, 2013, pp. 63–74. doi: 10.1145/2488222.2488260.

[171] *Crypto++ library 8.1*, https://www.cryptopp.com, Accessed on 20/05/2019.

[172] M. Sustrik *et al.*, *ZeroMQ*, https://github.com/zeromq/libzmq, Accessed on 31/07/2019.

[173] *Yahoo! finance*, https://finance.yahoo.com, Accessed on 20/05/2019.

[174] R. Silva, P. Barbosa and A. Brito, "Dynsgx: A privacy preserving toolset for dinamically loading functions into intel(r) sgx enclaves", in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2017, pp. 314–321. doi: 10.1109/CloudCom.2017.42.

[175] R. Ierusalimschy, L. H. de Figueiredo and W. C. Filho, "Lua — an extensible extension language", *Software: Practice and Experience*, vol. 26, no. 6, pp. 635–652, 1996. doi: 10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P.

[176] R. Ierusalimschy, *Programming in Lua, Fourth Edition*. Lua.Org, 2016.

[177] L. Leonini, E. Rivière and P. Felber, "SPLAY: Distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze)", in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*, Boston, Massachusetts, USA: USENIX Association, 2009, pp. 185–198. [Online]. Available: http://dl.acm.org/citation.cfm?id=1558977.1558990.

[178] C. F. Bolz and L. Tratt, "The impact of meta-tracing on vm design and implementation", *Science of Computer Programming*, vol. 98, no. P3, pp. 408–421, 2015. doi: 10.1016/j.scico.2013.02.001.

[179] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters", *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008. doi: 10.1145/1327452.1327492.

[180] P. Hintjens, *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, Inc., 2013.

[181] R. Barazzutti, T. Heinze, A. Martin, E. Onica, P. Felber, C. Fetzer, Z. Jerzak, M. Pasin and E. Rivière, "Elastic scaling of a high-throughput content-based publish/subscribe engine", in *2014 IEEE 34th International Conference on Distributed Computing Systems (ICDCS '14)*, 2014, pp. 567–576. doi: 10.1109/ICDCS.2014.64.

[182] J. MacQueen, "Some methods for classification and analysis of multivariate observations", in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, Berkeley, CA, USA: University of California Press, 1967, pp. 281–297.

[183] J. Daemen and V. Rijmen, *The Design of Rijndael*. Springer-Verlag New York, Inc., 2002. doi: 10.1007/978-3-662-04722-4.

[184] E. Curry, *Message-Oriented Middleware*. John Wiley & Sons, Ltd, 2005, pp. 1–28.

[185] T. Uustalu and V. Vene, "The essence of dataflow programming", in *Central European Functional Programming School*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 135–167. doi: 10.1007/11894100_5.

[186] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment", *Linux Journal*, vol. 2014, no. 239, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241.

[187] R. Bird and P. Wadler, *Introduction to Functional Programming*, English. Prentice Hall, 1988.

[188] *Docker Swarm*, https://www.docker.com/products/docker-swarm, Accessed on 20/05/2019.

[189] *ZeroMQ pipeline*, https://rfc.zeromq.org/30/, Accessed on 20/05/2019.

[190] *Docker Comopse*, https://docs.docker.com/compose/, Accessed on 12/03/2017.

[191] *Reactive Extensions for Lua*, https://github.com/bjornbytes/RxLua, Accessed on 20/05/2019.

[192] C. Szallies, "On using the observer design pattern", *XP-002323533,(Aug. 21, 1997)*, vol. 9, 1997. [Online]. Available: http://www.wohnklo.de/patterns/observer.html.

[193] *Lua binding to ZeroMQ*, https://github.com/zeromq/lzmq, Accessed on 20/05/2019.

[194] E. T. Bray, *The JavaScript object notation (JSON) data interchange format*, RFC 8259, 2017. doi: 10.17487/rfc8259. [Online]. Available: https://rfc-editor.org/rfc/rfc8259.txt.

[195] Y. Shafranovich, *Common format and MIME type for comma-separated values (CSV) files*, RFC 4180, 2005. doi: 10.17487/rfc4180.

[196] *Intel Core i7-6700*, http://ark.intel.com/products/88196/Intel-Core-i7-6700-Processor-8M-Cache-up-to-4_00-GHz, Accessed on 20/05/2019.

[197] *Consul*, http://consul.io, Accessed on 20/05/2019.

[198] *RITA | BTS*, http://www.transtats.bts.gov/OT_Delay/OT_DelayCause1.asp, Accessed on 20/05/2019.

[199] *Data Expo'09 ASA Statistics Computing and Graphics*, http://stat-computing.org/dataexpo/2009/the-data.html, Accessed on 20/05/2019.

[200] K. Webber. (2016). Diving into Akka streams, [Online]. Available: https://blog.redelastic.com/diving-into-akka-streams-2770b3aeabb0.

[201] E.-J. Goh, H. Shacham, N. Modadugu and D. Boneh, “SiRiUS: Securing remote untrusted storage”, in *Network and Distributed System Security Symposium (NDSS '03)*, vol. 3, 2003, pp. 131–145. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2017/09/SiRiUS-Securing-Remote-Untrusted-Storage-Eu-Jin-Goh.pdf.

[202] C. Delerablée, “Identity-based broadcast encryption with constant size ciphertexts and private keys”, in *Advances in Cryptology – ASIACRYPT 2007*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 200–215. doi: 10.1007/978-3-540-76900-2_12.

[203] B. Lynn *et al.* (2006). PBC library, [Online]. Available: https://crypto.stanford.edu/pbc/.

[204] T. Granlund *et al.*, *GMP, the GNU multiple precision arithmetic library*, 1991.

[205] T. Mayberry, E.-O. Blass and A. H. Chan, “Efficient private file retrieval by combining ORAM and PIR”, in *Network and Distributed System Security Symposium (NDSS '14)*, 2014. [Online]. Available: https://www.ndss-symposium.org/ndss2014/programme/efficient-private-file-retrieval-combining-oram-and-pir/.

[206] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi and D. Wichs, *Onion ORAM: A constant bandwidth blowup oblivious RAM*, Cryptology ePrint Archive, Report 2015/005, 2015. [Online]. Available: https://eprint.iacr.org/2015/005.

[207] D. Apon, J. Katz, E. Shi and A. Thiruvengadam, “Verifiable oblivious storage”, in *Public-Key Cryptography – PKC 2014*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 131–148. doi: 10.1007/978-3-642-54631-0_8.

[208] N. Ferguson and B. Schneier, *Practical cryptography*. Wiley New York, 2003, vol. 23.

[209] C. Ellison and B. Schneier, “Ten risks of PKI: What you're not being told about public key infrastructure”, *Computer Security Journal*, vol. 16, no. 1, pp. 1–7, 2000. [Online]. Available: https://www.schneier.com/academic/paperfiles/paper-pki.pdf.

[210] D. Boneh and M. Franklin, “Identity-based encryption from the Weil pairing”, in *Advances in Cryptology — CRYPTO 2001*, Berlin, Germany: Springer, 2001, pp. 213–229. doi: 10.1007/3-540-44647-8_13.

[211] B. Waters, “Efficient identity-based encryption without random oracles”, in *Advances in Cryptology – EUROCRYPT 2005*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 114–127. doi: 10.1007/11426639_7.

[212] C. Delerablée, P. Paillier and D. Pointcheval, “Fully collusion secure dynamic broadcast encryption with constant-size ciphertexts or decryption keys”, in *Pairing-Based Cryptography – Pairing 2007*, Berlin, Heidelberg, 2007, pp. 39–59. doi: 10.1007/978-3-540-73489-5_4.

[213] R. Sakai and J. Furukawa. (2007). Identity-based broadcast encryption, [Online]. Available: https://eprint.iacr.org/2007/217.

[214] Intel. (2017). Intel software guard extensions SSL, [Online]. Available: https://github.com/intel/intel-sgx-ssl.

[215] P. Schmidt *et al.* (2017). Linux kernel git revision history, [Online]. Available: https://www.kaggle.com/philschmidt/linux-kernel-git-revision-history.

[216] iDeals Virtual Data Rooms. (2019). Share and collaborate on business-critical documents in a secure way, [Online]. Available: https://www.idealsvdr.com/.

[217] D. Dolev and A. C. Yao, “On the security of public key protocols”, in *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science (SFCS '81)*, Washington, DC, USA: IEEE Computer Society, 1981, pp. 350–357. doi: 10.1109/SFCS.1981.32.

[218] N. Lohmann *et al.* (2018). JSON for modern C++, [Online]. Available: https://github.com/nlohmann/json.

[219] J. Han. (2017). OpenSSL library for SGX application, [Online]. Available: https://github.com/sparkly9399/SGX-OpenSSL.

[220] MongoDB Inc. (2019). MongoDB, [Online]. Available: https://www.mongodb.com/.

[221] ——, (2018). MongoDB C driver. version 1.12.0, [Online]. Available: http://mongoc.org/libmongoc/1.12.0/index.html.

[222] Minio, Inc. (2019). Minio: Private cloud storage, [Online]. Available: https://www.minio.io/.

[223] M. Backes, C. Cachin and A. Oprea, "Secure key-updating for lazy revocation", in *Computer Security – ESORICS 2006*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 327–346. doi: 10.1007/11863908_21.

[224] A. Shipilev *et al.* (2018). JMH: Java microbenchmark harness, Oracle Corporation, [Online]. Available: https://openjdk.java.net/projects/code-tools/jmh/.

[225] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan and R. Sears, "Benchmarking cloud serving systems with ycsb", in *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, Indianapolis, Indiana, USA: ACM, 2010, pp. 143–154. doi: 10.1145/1807128.1807152.

[226] E. Barker, W. Barker, W. Burr, W. Polk and M. Smid, "NIST special publication 800-57", *NIST Special publication*, vol. 800, no. 57, pp. 1–142, 2007.

[227] A. S. L. Special, https://www.somethingawful.com/weekend-web/aol-search-log/, 2007.

[228] A. S. D. S. U. P. to commit Murder, https://web.archive.org/web/20080605041024/http://plentyoffish.wordpress.com/2006/08/07/aol-search-data-shows-users-planning-to-commit-murder/, 2007.

[229] A. N. Langville and C. D. Meyer, *Google's PageRank and beyond: The science of search engine rankings*. Princeton University Press, 2011.

[230] A. Hannak, P. Sapiezynski, A. Molavi Kakhki, B. Krishnamurthy, D. Lazer, A. Mislove and C. Wilson, "Measuring personalization of web search", in *Proceedings of the 22Nd International Conference on World Wide Web (WWW '13)*, Rio de Janeiro, Brazil: ACM, 2013, pp. 527–538. doi: 10.1145/2488388.2488435.

[231] S. Yang and A. Ghose, "Analyzing the relationship between organic and sponsored search advertising: Positive, negative, or zero interdependence?", *Marketing Science*, vol. 29, no. 4, pp. 602–623, 2010. doi: 10.1287/mksc.1090.0552.

[232] C. Castelluccia, E. De Cristofaro and D. Perito, "Private information disclosure from web searches", in *Privacy Enhancing Technologies*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 38–55. doi: 10.1007/978-3-642-14527-8_3.

[233] H. Pang, J. Shen and R. Krishnan, "Privacy-preserving similarity-based text retrieval", *ACM Trans. Internet Technol.*, vol. 10, no. 1, 4:1–4:39, 2010. doi: 10.1145/1667067.1667071.

[234] Y. Lindell and E. Waisbard, "Private web search with malicious adversaries", in *Privacy Enhancing Technologies*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 220–235. doi: 10.1007/978-3-642-14527-8_13.

[235] L. Lamport, R. Shostak and M. Pease, "The byzantine generals problem", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982. doi: 10.1145/357172.357176.

[236] O. Goldreich, "Cryptography and cryptographic protocols", *Distrib. Comput.*, vol. 16, no. 2-3, pp. 177–199, 2003. doi: 10.1007/s00446-002-0077-1. [Online]. Available: http://dx.doi.org/10.1007/s00446-002-0077-1.

[237] A. Gervais, R. Shokri, A. Singla, S. Capkun and V. Lenders, "Quantifying web-search privacy", in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14, Scottsdale, Arizona, USA: ACM, 2014, pp. 966–977. doi: 10.1145/2660267.2660367.

[238] *Wrk2*. https://github.com/giltene/wrk2, 2015.

[239] J. Seward, N. Nethercote and J. Weidendorfer, *Valgrind 3.3 - Advanced Debugging and Profiling for GNU/Linux Applications*. Network Theory Ltd., 2008.

[240] J. W. Palmer, "Web Site Usability, Design, and Performance Metrics", *Information Systems Research*, vol. 13, no. 2, pp. 151–167, 2002. doi: 10.1287/isre.13.2.151.88.

[241] D. Goltzsche, S. Rüsch, M. Nieke, S. Vaucher, N. Weichbrodt, V. Schiavoni, P. Aublin, P. Cosa, C. Fetzer, P. Felber, P. Pietzuch and R. Kapitza, "Endbox: Scalable middlebox functions using client-side trusted execution", in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Luxembourg, 2018, pp. 386–397. doi: 10.1109/DSN.2018.00048.

[242] M. Pease, R. Shostak and L. Lamport, "Reaching agreement in the presence of faults", *Journal of the ACM (JACM)*, vol. 27, no. 2, pp. 228–234, 1980. doi: 10.1145/322186.322188.

[243] Google group, *Google privacy & terms*, https://policies.google.com/privacy/key-terms?hl=en, Accessed on 30/07/2019.

[244] C. Fellbaum, Ed., *WordNet: an electronic lexical database*. MIT Press, 1998. [Online]. Available: https://mitpress.mit.edu/books/wordnet.

[245] A. González, G. Rigau and M. Castillo, "A graph-based method to improve WordNet domains", in *Computational Linguistics and Intelligent Text Processing*, Springer, 2012, pp. 17–28. doi: 10.1007/978-3-642-28604-9_2.

[246] D. M. Blei, A. Y. Ng and M. I. Jordan, "Latent Dirichlet allocation", *Journal of Machine Learning Research*, vol. 3, pp. 993–1022, 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=944919.944937.

[247] A. K. McCallum, *MALLET: A machine learning for language toolkit*, Accessed on 30/07/2019. [Online]. Available: http://mallet.cs.umass.edu.

[248] A. Mazieres, M. Trachman, J.-P. Cointet, B. Coulmont and C. Prieur, "Deep tags: Toward a quantitative analysis of online pornography", *Porn Studies*, vol. 1, no. 1, pp. 80–95, 2014.

[249] Google Trends, https://trends.google.com, 2017.

[250] P. Hintjens *et al.*, *Zyre - an open-source framework for proximity-based peer-to-peer applications*, https://github.com/zeromq/zyre, Accessed on 31/07/2019.

[251] F. Zhang, *mbedtls-SGX: A SGX-friendly TLS stack (ported from mbedtls)*, 2017. [Online]. Available: https://github.com/bl4ck5un/mbedtls-SGX.

[252] CrowdFlower, http://www.crowdflower.com, 2017.

[253] A. Kumar, A. Kashyap, V. Phegade and J. Schrater, "Self-Defending Key Management Service with Intel Software Guard Extensions", Fortanix, White Paper, 2018. [Online]. Available: https://www.fortanix.com/assets/SGXwhitepaper/Fortanix_SDKMS_with_Intel_SGX_Whitepaper.pdf.

[254] S. Chakrabarti, B. Baker and M. Vij, "Intel SGX enabled key manager service with openstack barbican", *arXiv e-prints*, 2017. [Online]. Available: https://arxiv.org/pdf/1712.07694.

[255] A. Kosba, A. Miller, E. Shi, Z. Wen and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts", in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 839–858. doi: 10.1109/SP.2016.55.

[256] H. Dang, D. Le Tien and E.-C. Chang, "Towards a marketplace for secure outsourced computations", in *Computer Security – ESORICS 2019*, Cham: Springer, 2019, pp. 790–808. doi: 10.1007/978-3-030-29959-0_38.