unine

Université de Neuchâtel
Institut d'informatique

Exploring the potential of modern hardware-assisted security and networking technologies

par

Sébastien Vaucher

Thèse

présentée à la Faculté des sciences de l'Université de Neuchâtel
pour l'obtention du grade de Docteur ès sciences

Acceptée sur proposition du jury :

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

Dr Valerio Schiavoni
Université de Neuchâtel, Suisse

Prof. Alexandra Fedorova
The University of British Columbia, Canada

Dr Sonia Ben Mokhtar
CNRS / INSA Lyon, France

Prof. Gaël Thomas
Télécom SudParis, France

Soutenue le 10 octobre 2022

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 Sébastien VAUCHER

Titre:

“Exploring the potential of modern hardware-assisted security and networking technologies”

sur le rapport des membres du jury composé comme suit:

- Prof. Pascal Felber, directeur de thèse, Université de Neuchâtel, Suisse
- Dr Valerio Schiavoni, Université de Neuchâtel, Suisse
- Prof. Alexandra Fedorova, Uni. British Columbia, Canada
- Dr Sonia Ben Mokhtar, CNRS/INSA, Lyon, France
- Prof. Gaël Thomas, Télécom SudParis, France

Neuchâtel, le 8 novembre 2022

Le Doyen, Prof. R. Bshary

Abstract

Companies progressively migrate their computing workloads to the cloud to cope with growing computing needs and simplify their day-to-day operations. On the tenants' side, offloading computations to a third-party can pose privacy and security issues due to the loss of control over the data. At the same time, cloud operators need to cope with ever-increasing amounts of network traffic.

New hardware-assisted technologies such as trusted execution environments (TEEs) and data plane-programmable switches represent promising innovations. The former allows to offload computations on sensitive data to an untrusted third-party in a privacy-preserving way. Programmable switches similarly represent a breakthrough in hardware networking appliances, enabling full programmability with hardly any performance trade-offs.

In this thesis, we investigate how these hardware-assisted technologies can help solve contemporary problems. We present various new systems to show that hardware-assisted mechanisms not only allow to perform existing tasks more efficiently, but also that completely new solutions are made possible.

First, we evaluate the performance of a couple of commercial TEEs, in part by using tools that we build. We find out that the superior security guarantees offered by Intel Software Guard Extensions (SGX) compared to AMD Secure Encrypted Virtualization (SEV) lead to usually worse performance, but that clever use of the technology can avoid those overheads. Nonetheless, we discover that the performance of a particular TEE can change over time as security fixes and mitigations are released by its vendor.

As Intel SGX 1 is rather limited in regards to memory usage, we design and develop an orchestrator that allows to efficiently deploy distributed containerized SGX workloads, while equally supporting legacy workloads and servers (without SGX).

Thereupon, we solve a common problem which is how to share data among a group in a privacy-preserving way. We design and develop a scalable cloud-hosted system that improves upon the state-of-the-art by three orders of magnitude, thanks to a reduction in complexity of a cryptographic algorithm permitted by the security guarantees given by SGX.

Subsequently, we show that having SGX on client machines allows to offload workloads traditionally performed by dedicated network appliances to clients. With this unconventional arrangement, unused client resources can help large network operators in coping with increased network usage. Our prototype scales linearly with the number of clients, achieving up to 3.8× higher throughput than a similar centralized network appliance.

In keeping with finding solutions to cope with the ever increasing amount of network traffic, we show that the generalized deduplication (GD) technique can be used as a compression algorithm. Thanks to its limited use of resources and constant-time execution, we can implement a prototype on top of a programmable switch. We can hence transparently compress and decompress network packets within the network itself, without any additional pieces of hardware, and with no performance penalties. Our prototype manages to reduce the size of a real-world data trace by 90 %.

Finally, we perform in-network disaggregation of non-volatile main memory (NVMM), in line with the contemporary trend of *hyperconvergence*. The versatility and location of programmable switches within the network allow us to develop a prototype that transparently intercepts and modifies remote direct memory access (RDMA) connections between clients and NVMM servers. Using the stateful capabilities of the programmable switch, we implement a data striping and mirroring mechanism that provides faster-than-native accesses to NVMM while increasing reliability.

Keywords: Trusted execution environments, Data-plane programmable switches, Container orchestration, Network security, Network protocols.

Résumé

De plus en plus d'entreprises migrent progressivement leurs charges de travail informatiques vers le *cloud* afin de répondre à leurs besoins croissants en ressources informatiques tout en simplifiant leurs opérations quotidiennes. Transférer la responsabilité de l'infrastructure à un tiers peut poser des problèmes de confidentialité et de sécurité en raison de la perte de contrôle par rapport aux données. Concurremment, les opérateurs de services *cloud* doivent faire face à un trafic réseau en constante augmentation.

Les nouvelles technologies d'accélération matérielle, telles que les environnements d'exécution de confiance (TEE) et les commutateurs programmables au niveau du plan de données, représentent des innovations prometteuses. La première permet de décharger des calculs concernant des données sensibles vers un tiers auquel on ne fait pas confiance tout en préservant la confidentialité des données. Les commutateurs programmables représentent quand à eux une avancée dans le domaine des appareils de mise en réseau, permettant une programmabilité complète sans pour autant compromettre la performance.

Dans cette thèse, nous étudions comment ces deux technologies d'accélération matérielle peuvent aider à résoudre des problèmes contemporains. Nous présentons plusieurs nouveaux systèmes afin de montrer que l'accélération matérielle permet non seulement d'exécuter des tâches existantes plus efficacement, mais aussi qu'elle permet de créer des solutions complètement nouvelles.

Nous commençons par évaluer les performances de quelques TEE commerciaux, en partie en construisant nos propres outils. Nous constatons que les garanties de sécurité supérieures offertes par la technologie Software Guard Extensions (SGX) d'Intel par rapport à la technologie Secure Encrypted Virtualization (SEV) d'AMD conduisent à des performances généralement inférieures, mais qu'une utilisation intelligente de la technologie permet d'éviter ces surcoûts. Néanmoins, nous découvrons que les performances d'un TEE en particulier peuvent changer au fil du temps, à mesure que les correctifs de sécurité sont publiés par son fournisseur.

Comme Intel SGX 1 possède des limites importantes quant à l'utilisation mémoire, nous concevons et développons un orchestrateur qui permet de déployer efficacement des charges de travail SGX conteneurisées de manière répartie, tout en intégrant les charges de travail et serveurs existants (sans SGX).

Par la suite, nous résolvons un problème fréquent, à savoir comment partager des données au sein d'un groupe tout en préservant la confidentialité. Nous concevons et développons un système élastique hébergé dans le *cloud* dont la performance surpasse l'état de l'art de trois ordres de grandeur, grâce à une réduction de la complexité d'un algorithme cryptographique permise par les garanties de sécurité que SGX fournit.

Ensuite, nous montrons qu'avoir SGX sur les machines clientes permet d'y décharger des charges de travail traditionnellement effectuées par des appareils réseau dédiés. Grâce à cet arrangement non conventionnel, les ressources inutilisées des machines clientes peuvent aider les grands opérateurs de réseau à faire face à l'utilisation accrue du réseau. Notre prototype passe à l'échelle de façon linéaire en fonction du nombre de clients, atteignant un débit jusqu'à 3.8× supérieur à celui d'un appareil réseau centralisé similaire.

Toujours dans le but de trouver des solutions permettant de faire face à l'augmentation constante du trafic réseau, nous montrons que la technique de dé-duplication généralisée (GD) peut être utilisée comme algorithme de compression. Grâce à son utilisation restreinte en termes de ressources et à son exécution en temps constant, nous pouvons implémenter un prototype intégré à un commutateur programmable. Nous pouvons donc compresser et décompresser de manière transparente les paquets réseau au sein même du réseau, sans matériel supplémentaire et sans perte de performance. Notre prototype parvient à réduire la taille d'une trace de données provenant du monde réel de près de 90 %.

Enfin, nous proposons de désagréger la mémoire de travail non-volatile (NVMM) dans le réseau, conformément à la tendance contemporaine d'*hyper-convergence*. La polyvalence et la position des commutateurs programmables dans le réseau nous permettent de développer un prototype qui intercepte et modifie de manière transparente les accès directs à la mémoire distante (RDMA) entre les clients et les serveurs NVMM. Nous tirons parti de la présence de registres dans le commutateur programmable pour implémenter un mécanisme d'entrelacement et de mise en miroir des données qui permet d'accéder à la NVMM plus rapidement qu'en mode natif tout en augmentant la fiabilité.

Mots-clés : Environnements d'exécution de confiance, Commutateurs programmables au niveau du plan de données, Orchestration de conteneurs, Sécurité des réseaux, Protocoles réseau.

Remerciements

J'adresse mes plus sincères remerciements à mon directeur de thèse, Prof. Pascal Felber, tout d'abord pour m'avoir donné l'opportunité d'effectuer ce doctorat, puis pour son incontournable soutien pendant celui-ci. Sa capacité à toujours immédiatement trouver une solution à tout problème quel que soit sa nature ou sa complexité s'est avérée indispensable à de nombreuses reprises.

Je remercie tout particulièrement Dr Valerio Schiavoni pour sa collaboration de longue date, de la co-direction de ma thèse de Master, aux discussions fructueuses que nous avons eues pendant le doctorat, aux inestimables heures qu'il a passées à l'écriture et la relecture des articles dont nous sommes co-auteurs, pour terminer à sa participation au jury de thèse.

I would like to express my deepest appreciation to the external members of the jury, Prof. Alexandra Fedorova, Dr Sonia Ben Mokhtar, and Prof. Gaël Thomas for the time they dedicated to the thorough evaluation of this manuscript, and for the in-depth discussions we had during the defense.

Merci à mon collègue de bureau historique et surtout ami Dr Rafael Pires. Ses vastes connaissances techniques m'ont permis d'entrer dans le sujet en un rien de temps. Il m'a surtout appris à déjouer les pièges du monde académique, ce qui nous a permis d'efficacement co-écrire un nombre respectable d'articles scientifiques.

Many thanks to the collaborators of Cisco Switzerland in Écublens, in particular to Dr Patrick Marlier and Eric Rose, who made my internship there a success despite the difficulties resulting from the pandemic.

I am also thankful to the researchers from Aarhus Universitet, Université de Bordeaux, Technische Universität Braunschweig, Technische Universität Dresden, Imperial College London, Microsoft Research, and Télécom SudParis with whom I collaborated on several research efforts during my thesis.

Je remercie Dr Dorian Burihabwa, Rémi Dulong et Jämes Ménétrey, avec qui j'ai accidentellement testé la résistance à l'écrasement d'une fibre optique multimode, et plus généralement partagé les tâches d'administration de l'infrastructure de calcul sur laquelle les membres de l'institut reposent pour leurs expériences scientifiques. Grâce à leur travail, le *cluster* est actuellement dans le meilleur des états.

Un grand merci aux vingt-sept doctorantes et doctorants de l'institut avec qui j'ai partagé cette aventure. J'ai grandement apprécié de pouvoir discuter de tout et rien avec vous chaque midi au *saloon*, m'apportant l'énergie nécessaire à l'accomplissement de ma thèse. J'ai encore plus apprécié de pouvoir partager de bons moments avec vous lors des écoles doctorales ou hors de tout cadre universitaire.

Je tiens donc à notamment remercier Jämes Ménétrey, Laurent Hayez, Maria Carpen-Amarie et Mirco Kocher d'avoir partagé, et tantôt instigué ces bons moments. *Gracias a* Raziel Carvajal

Gómez de m’avoir fait découvrir que les authentiques spécialités culinaires mexicaines sont tout autant délicieuses qu’elles sont épicées. *Grazie a* Roberta Barbi de nous avoir apporté de si délicieux gâteaux faits maison à toute occasion. *Vielen Dank an* Christian Göttel de s’être occupé du *saloon* à tel point que mon rôle de suppléant est devenu complètement honorifique. *Thank you* Peterson Yuhala d’être toujours prêt à tenter de nouvelles expériences.

Merci à Dorian Burihabwa de nous avoir si souvent spontanément proposé d’aller boire un verre (ou deux) après le travail. Ces sorties sont indubitablement le socle de la bonne ambiance qui règne à l’institut.

Merci à Rémi Dulong pour sa présence virtuelle tout au long des périodes de télétravail obligatoire. C’est toujours un plaisir de camionner virtuellement avec lui, ou d’effectuer un vol en autogire virtuel, ou de jouer moult parties de football en voitures-fusées. Mais c’est encore plus volontiers que je partage avec lui une vraie fondue au fromage sous un orage et une pluie battante bien réels.

Enfin, *obrigado* Rafael Pires pour les innombrables activités et repas que nous avons partagés ensemble. S’il existe une carte de fidélité qui les compte, je ne sais combien d’exemplaires seraient remplis de *stempfs*.

Pour terminer, mes plus chaleureuses accolades vont aux personnes qui comptent le plus à mes yeux, mes parents Donatella et Jean-Marc, ainsi que ma sœur Marion, sans qui rien de ce que je suis devenu n’aurait été ne serait-ce qu’envisageable.

Contents

List of acronyms

AE	authenticated encryption
AES	advanced encryption standard
AESM	application enclave service manager
ALU	arithmetic–logic unit
ANOBE	anonymous broadcast encryption
API	application programming interface
APM	appliance persistency method
ASIC	application-specific integrated circuit
AWS	Amazon Web Services
BfRt	Barefoot runtime
BIOS	basic input/output system
BYOD	bring your own device
CA	certificate authority
CCX	core complex
CDF	cumulative density function
CPU	central processing unit
CRC	cyclic redundancy check
DC	data center
DDIO	Intel Data Direct I/O
DDoS	distributed denial-of-service
DH	Diffie-Hellman
DNS	domain name system
DoS	denial-of-service
DPI	deep packet inspection
DS	differentiated services
EC	erasure code
EC2	Amazon Elastic Compute Cloud
ECDH	elliptic curve Diffie–Hellman
ECDSA	elliptic curve digital signature algorithm
EPC	enclave page cache
EPID	Intel Enhanced Privacy ID
FCFS	first come first serve
FPGA	field-programmable gate array
GCM	Galois counter mode
GD	generalized deduplication
GnuPG	GNU Privacy Guard
GPU	graphics processing unit
gRPC	Google remote procedure calls
HMAC	keyed-hash message authentication code
HTTP	hypertext transfer protocol
IaaS	infrastructure as a service
IAS	Intel Attestation Service
ICMP	Internet control message protocol
ICRC	invariant CRC
IDPS	intrusion detection and prevention system
IoT	Internet of things
IP	Internet protocol
ISP	Internet service provider
IT	information technology
IV	initialization vector
JMH	Java Microbenchmark Harness
JSON	JavaScript object notation
LE	launch enclave
LRU	least recently used
M&A	merger and acquisition
MAC	message authentication code
MEE	memory encryption engine
MitM	man-in-the-middle
MSB	most significant bit
MTU	maximum transmission unit
NFV	network functions virtualization
NPL	network programming language
NPU	network processing unit
NVMM	non-volatile main memory
OoM	order of magnitude
OS	operating system
PE	provisioning enclave
PGP	Pretty Good Privacy
PID	process identifier
PMem	Intel Optane Persistent Memory
PRM	processor reserved memory
PSA	portable switch architecture
PSN	packet sequence number
PSW	platform software
QE	quoting enclave

RAID	redundant array of independent disks
RAM	random access memory
RDMA	remote direct memory access
REST	representational state transfer
RMW	read-modify-write
RoCE	RDMA over Converged Ethernet
ROM	read-only memory
RTT	round-trip time
S3	Amazon Simple Storage Service
SDE	software development environment
SDK	software development kit
SDN	software-defined networking
SEV	AMD Secure Encrypted Virtualization
SEV-ES	AMD SEV Encrypted State
SEV-SNP	AMD SEV Secure Nested Paging
SGX	Intel Software Guard Extensions
SHA	secure hash algorithm
SME	AMD Secure Memory Encryption
SMM	system management mode
SoC	system-on-chip
SRAM	static random access memory
SSD	solid state drive
TA	trusted authority
TCB	trusted computing base
TCP	transmission control protocol
TEE	trusted execution environment
TLS	transport layer security
TNA	Intel Tofino Native Architecture
TPM	trusted platform module
TTL	time to live
UDP	user datagram protocol
UEFI	unified extensible firmware interface
VA	virtual address
VDR	virtual data room
VM	virtual machine
VPN	virtual private network
YCSB	Yahoo! Cloud Serving Benchmark

List of figures

List of tables

List of algorithms

Chapter 1.

Introduction

1.1. Context and motivation

Connected computing devices are omnipresent. Most of them consume data provided by applications running on remote servers. Accordingly, the amount of worldwide network traffic increases by 30 % every year [95]. Companies offering Internet-facing services regularly need to increase the computing capabilities of their servers. Instead of buying, configuring and maintaining physical servers by themselves, companies increasingly turn to cloud computing to offload their workloads [50]. While some of the burden gets correspondingly offloaded to the cloud provider, other issues arise due to this delegation of responsibilities. Security and privacy issues in particular have been the center of attention, particularly when large entities have suffered data leakages. On the other hand, economical factors dictate that cloud providers need to use their infrastructure as efficiently as possible.

Trusted execution environments (TEEs) represent a promising innovation that allows entities to offload their computations to an external entity without having to completely trust it. When used correctly, the technology also prevents malicious actors from mounting an attack against the offloaded service. Within the realm of the x86 processor architecture, Intel Software Guard Extensions (SGX) and AMD Secure Encrypted Virtualization (SEV) are widely supported on server-grade processors sold by their respective vendors.

The steady increase in network traffic also poses another serious challenge to cloud providers. Ossified network protocols and unmodifiable hardware appliances require them to anticipate future needs, and constantly replace older models with new ones to support improved networking techniques. Software-defined networking (SDN) allows operators of large networks to programmatically manage their network in order to regulate network flows in real-time. However, SDN only deals with the routing process (*control plane*), and stays away from the forwarding process itself (*data plane*). In this context, another promising innovation are data plane-programmable switches, which enable complete customization of the forwarding process. White-box programmability allows researchers to experiment with novel networking techniques and protocols without having to purchase new hardware. Remarkably, the performance of those appliances, notably the Intel Tofino family of switch application-specific integrated circuits (ASICs), matches or even surpasses fixed-function ones.

Both TEEs and programmable switches provide enhanced capabilities thanks to specialized hardware. Developing software for these targets requires to keep their particularities in mind to glean their full potential.

In this work, we showcase how cloud providers and other large network operators can leverage these new hardware-assisted technologies to help alleviate issues that relate to privacy, security, and networking. Before doing so, we evaluate Intel SGX and AMD SEV TEEs to understand their security guarantees and behavior under load. Similarly, we develop a complete framework around a container orchestration engine such that we could seamlessly use Intel SGX in a distributed way. Using our findings and the orchestration framework, we show how we can build a scalable cloud-hosted privacy-preserving data sharing system. Thanks to our use of TEEs, we can reduce the complexity of an established cryptographic scheme by several orders of magnitude. Shifting to networking issues, we show how, instead of offloading computations to large entities, network operators can securely employ underused end-user resources to perform packet filtering and other security-related tasks. Continuing, we show that programmable switches are capable of performing in-network compression of packets at line rate, potentially reducing bandwidth across large networks. Finally, we extend the technique of *hyperconvergence* to non-volatile main memory (NVMM), while transparently providing faster-than-native throughput and better reliability by using a programmable switch to intercept and transform data transfer packets.

1.2. Contributions

The contributions of this thesis are the following.

- We extend a well-known *stressing* utility that can stress Intel SGX enclaves. In addition, our construction, **STRESS-SGX**, doubles as a benchmarking tool. We openly release our contribution for the benefit of the community. We discover uncommunicated performance overheads introduced by a microcode patch.
  Work on STRESS-SGX [175] was supervised by Valerio Schiavoni and Pascal Felber. It constitutes sections 4.2, 4.3.2 and 4.3.4 of this manuscript.
- We evaluate and compare the respective characteristics and performance of competing TEEs for the x86 architecture: Intel SGX and AMD SEV. We use STRESS-SGX as well as other benchmarking tools.
  This work [64] was done in collaboration with Christian Göttel, Rafael Pires and Isabelly Rocha, and supervised by Pascal Felber, Marcelo Pasin and Valerio Schiavoni. Results span sections 2.2, 4.3.3 and 4.4.
- We design and implement an orchestrator for Intel SGX-enabled containerized workloads running on heterogeneous clusters containing SGX-enabled machines as well as legacy ones. We openly release **SK8S** to the community.
  This research effort [173] was realized in collaboration with Rafael Pires, and supervised by Pascal Felber, Marcelo Pasin, Valerio Schiavoni and Christof Fetzer (Technische Universität Dresden). The contribution constitutes chapter 5 of this thesis, while an SGX-specific result is presented in section 4.3.1.
- We discover that, thanks to the trusted execution environment provided by SGX, it is possible to greatly simplify a cryptographic algorithm for group data sharing. We design

and implement **A-Sky**, a scalable system based on containerized micro-services. We openly release our contribution [37] to the community.
It results from a collaboration with Stefan Contiu (Université de Bordeaux) and Rafael Pires, supervised by Marcelo Pasin, Pascal Felber and Laurent Réveillère (Université de Bordeaux). A-Sky is described in details in chapter 6.

- Departing from the practice of offloading workloads to cloud infrastructures, we offload traditionally centrally-executed network security function to underused client machines using Intel SGX as the root of trust.
  This research effort [61] was led by David Goltzsche, Signe Rüsch and Manuel Nieke, in collaboration with Nico Weichbrodt (all from Technische Universität Braunschweig), Valerio Schiavoni, Pierre-Louis Aublin (Imperial College London) and Paolo Costa (Microsoft Research). Christof Fetzer (Technische Universität Dresden), Pascal Felber, Peter Pietzuch (Imperial College London) and Rüdiger Kapitza (Technische Universität Braunschweig) supervised the work. The resulting system, **EndBox**, is presented in chapter 7.

- Using the generalized deduplication (GD) technique, we devise a compression algorithm that fits the resources of a programmable switch. Our implementation, **ZipLine**, consequently runs transparently at line rate within a network.
  ZipLine [176] was a collaboration with Niloofar Yazdani (Aarhus Universitet), with Pascal Felber, Daniel Lucani (Aarhus Universitet) and Valerio Schiavoni supervising. We explain our construction in chapter 8.

- Considering the emergence of NVMM and *hyperconvergence*, we create a transparent in-network system to disaggregate NVMM servers while increasing data reliability at faster-than-native speeds.
  NVSplit is yet-unpublished work jointly realized with Rémi Dulong under the supervision of Pascal Felber and Gaël Thomas (Télécom SudParis). Chapter 9 explains how the system works.

Another two articles [36, 65] were published during this thesis, but they do not form part of this manuscript. The complete bibliographic reference of each of our publications is listed starting on page 109.

1.3. Outline

The rest of this manuscript is organized as such.

Part I explains the two pieces of technology that we use throughout our work: TEEs in chapter 2 and programmable switches in chapter 3.

In **part II**, we present our own research related to TEEs. We start by evaluating Intel SGX and AMD SEV in chapter 4. Then, we present our SGX-aware container orchestrator in chapter 5. We introduce the A-Sky anonymous group data sharing system in chapter 6. We conclude the part by explaining the EndBox client-side middleboxes system in chapter 7.

Our research that revolves around programmable switches is contained in **part III**. In chapter 8, we expose ZipLine, an in-network data compression system. We terminate by presenting NVSplit, an NVMM disaggregator in chapter 9.

Finally, we conclude the thesis in chapter 10.

Part I.

Background

Chapter 2.

Trusted execution environments

Running code in the cloud requires to unconditionally trust the cloud provider and its infrastructure. There exist no tangible guarantees that the code and data are correctly processed. To overcome these issues, one could adopt solutions based on homomorphic cryptosystems [58]. However, their performance is several orders of magnitude slower than native systems, and as of today, these solutions are still impractical for real-world deployment and adoption [59].

Trusted execution environments (TEEs) bring a hardware solution to the problem by shielding the processing in such a way that even a physical attack is unlikely to affect the integrity and confidentiality of the data. Thanks to its availability on a large range of commercially-sold processors, Intel Software Guard Extensions (SGX) has become a popular TEE for the x86 architecture. Another TEE for the x86 processor architecture is AMD Secure Encrypted Virtualization (SEV). We introduce and compare these two TEEs in subsequent sections.

TEEs are also commercially available on other processor architectures. Most notably, TrustZone [4] is widely available and used on ARM-based system-on-chips (SoCs). MultiZone Security [77] is a TEE for the RISC-V architecture.

2.1. Intel Software Guard Extensions

Intel SGX is a TEE that is available on Intel processors derived from the Skylake architecture. Processors up to the Comet Lake generation understand the original SGX 1 set of instructions, while more recent Ice Lake server-grade processors support SGX 2.[1]

SGX-enabled applications create secure *enclaves* that protect the integrity and confidentiality of the code being executed and its associated data. SGX protects against software-based attacks, but also against more-involved physical attacks. The central processing unit (CPU) package is the security boundary: it is assumed that uncapping and prying into a powered-on CPU is infeasible. However, other components are not part of the trusted computing base (TCB).

Data relative to an SGX enclave may be stored in one of three memory areas [40]. Within the CPU itself (data in registers or caches), hardware access control mechanisms ensure that other processes, regardless of their privileges, cannot access or tamper with an SGX enclave. SGX enclaves can also use a dedicated subset of system memory called the enclave page cache (EPC). The EPC is split in 4 KiB pages and exists in processor reserved memory (PRM), a range of system memory that is inaccessible to other programs running on the same machine,

[1] Intel also refers to SGX 2 as Enclave Dynamic Memory Management (EDMM).

including privileged software such as the operating system. Data integrity is safeguarded by associating metadata that is itself integrity-protected. The metadata is stored in a Merkle tree structure [56], the root of which is stored in static random access memory (SRAM) within the CPU package.

Processors that support SGX 1 can use up to 128 MiB of EPC, out of which 93.5 MiB can effectively be used by applications, while the rest is used for storing SGX metadata. The effective size of the EPC is configurable using unified extensible firmware interface (UEFI) parameters. Note that the EPC is shared among all the applications executing inside enclaves, hence being a very scarce and highly contended resource.

In order to overcome the size limitation of the EPC, SGX implements a paging mechanism. It can page-out portions of trusted memory into regular system memory. An access to an enclave page that does not reside in the EPC triggers a page fault. The SGX kernel module interacts with the CPU to choose the pages to evict. The memory encryption engine (MEE) [68] ensures that data moving between the CPU and main memory stays confidential. It features encryption, tamper resistance, and replay protection mechanisms. If a cache miss hits a protected region, the MEE encrypts or decrypts data before sending to, respectively fetching from, the system memory and performs integrity checks. Thanks to these mechanisms, a memory dump of an SGX enclave will only produce unusable encrypted data. The enclave code is contained within a signed, but unencrypted shared library, meaning that it can be inspected by potential attackers, unlike its associated data.

An SGX enclave can be remotely *attested*. One can verify that a remote machine runs the intended code on a genuine Intel processor with SGX enabled. There are two methods available [94]: (i) Intel Enhanced Privacy ID (EPID), available on "selected" SGX 1 processors, and (ii) elliptic curve digital signature algorithm (ECDSA) attestation, available on processors supporting SGX 2. With the former, the attestation happens online through Intel Attestation Service (IAS). A business agreement with Intel is necessary to use it. In the latter case, the attestation process can be entirely under the control of the infrastructure provider.

Data can be saved on persistent storage using a *seal key* that is specific to one particular program running on a particular processor. With this mechanism, one can store certificates in order to waive the need to perform a remote attestation procedure every time an enclave application restarts.

The execution flow of a program using SGX enclaves is represented in figure 2.1. First, the *untrusted* part of the application creates an enclave ❶. The enclave must be initialized using a *launch token* given by the launch enclave (LE) that Intel provides. Access to the LE and other *architectural enclaves*, such as the quoting enclave (QE) and the provisioning enclave (PE) is provided by Intel application enclave service manager (AESM). To call a trusted function ❷, and thus enter the enclave, the program performs an *ecall* ❸. The program goes through the enclave call gate to bring the execution flow in SGX-protected mode ❹. One of the thread of the enclave executes the trusted function ❺. When the function returns, its result is sent back (potentially encrypted) ❻ before giving control back to the initial thread ❼.

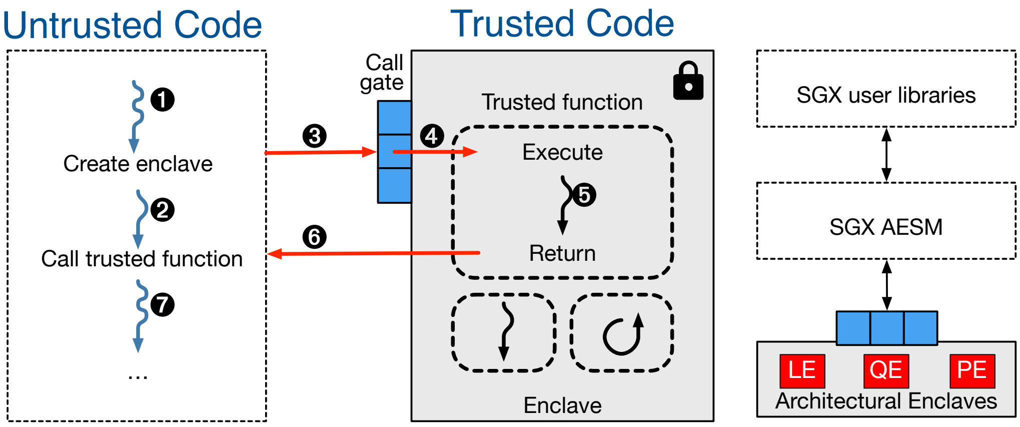

Figure 2.1.: SGX enclave execution flow.

2.1.1. Microcode updates

With the exception of the MEE, SGX is entirely implemented in *microcode* [40]. In section 4.3.2, we show that the performance of SGX depends on the version of the microcode that a CPU uses. Due to the complexity of the x86 architecture, with its abundance of instructions [88], it is advantageous to implement certain instructions in *microcode* instead of pure hardware.

An initial version of the microcode is contained in read-only memory (ROM) on every Intel processor. Starting with the P6 microarchitecture introduced in 1995, the microcode can be updated by end users [69]. The update process is transient: the factory-shipped version of the microcode stays unchanged in ROM, with the patch only overlaying the parts of the original microcode that need fixing. Microcode updates are applied at boot time, first by the basic input/output system (BIOS) or UEFI. The operating system (OS) can also apply a microcode update during its early booting sequence. Intel provides microcode updates in the form of opaque encrypted files. Processors only accept microcode updates signed by Intel, and with a higher version code than the currently-loaded microcode (rollback protection).

2.2. AMD Secure Encrypted Virtualization

Similarly to Intel, its main competitor AMD introduced its own TEE technology: SEV [103]. It has been brought to market in mid-2017 together with the *AMD EPYC* family of server processors, based on the *Zen* processor micro-architecture [1, 119]. It provides transparent encryption of the memory used by virtual machines (VMs). To use this technology, the Secure Memory Encryption (SME) extension must be available and supported by the underlying hardware. SME encrypts the entire memory using a single key. To limit overheads, the architecture relies on a hardware advanced encryption standard (AES) engine embedded within the memory controller. SEV improves upon SME by generating a distinct key for each VM. The creation of ephemeral encryption keys is delegated to the *secure processor*, an ARM TrustZone-enabled on-die SoC [103]. The keys are never exposed to software executed by the CPU itself.

Table 2.1.: Comparison of the TCB of different TEEs.

	SGX 1	SEV	SEV-ES
Other VMs	X	X	X
Hypervisor	X	✓	X
Host OS	X	✓	X
Guest OS	X	✓	✓
Privileged user	X	✓	✓
Untrusted code	X	✓	✓
Trusted code	✓	✓	✓

Table 2.2.: Comparison of the security guarantees and memory limits of different TEEs.

	SGX 1	SEV
Integrity	✓	X
Freshness	✓	X
Encryption	✓	✓
Memory limit	93.5 MiB	*none*

From a programmer perspective, SEV is completely transparent, unlike Intel SGX. Hence, the execution flow of a program using SEV is the same as a regular program. It is possible to attest encrypted states by using an internal challenge mechanism, so that a program can receive proof that a page is being correctly encrypted.

SEV Encrypted State (SEV-ES) [102] is an extension of SEV that additionally protects the execution and register state of an entire VM from a compromised hypervisor, host OS or co-hosted VM. Unmodified applications are hence protected against attackers with full control over the hosting machine, as it can only access encrypted memory pages.

We summarize the differences in terms of TCB, and security guarantees and limitation between Intel SGX, AMD SEV and SEV-ES in tables 2.1 and 2.2, respectively.

Chapter 3.

Programmable networks

Traditionally, vendors would market a range of network appliances, each supporting a specific set of features and standardized network protocols. Programmable switches represent a new paradigm where the low-level switching processor (the *data plane*) becomes fully programmable. It allows network engineers to have fine-grain control on packet flows, as well as execute custom operations on those packets, effectively leading to *in-network* processing. Further, future or custom network protocols can be implemented as well as standardized ones. Whereas field-programmable gate arrays (FPGAs) or fast software-based network libraries (*e.g.*, DPDK [120]) could already cover such use-cases, programmable switches do so at line-rate across all ports of the appliance.

3.1. The P4 language

Data plane programming is achieved using purpose-specific programming languages. The P4 language [28] is becoming the *de facto* standard supported by switch application-specific integrated circuits (ASICs) with programmable data planes.[1] In fact, P4 supports multiple targets beyond ASICs: FPGAs, network processing units (NPUs), and pure software targets [75].

There exist two incompatible variants of P4: $P4_{14}$ and $P4_{16}$—first released in 2014 and 2016, respectively. $P4_{14}$ is considered deprecated; our work therefore only uses $P4_{16}$ and any mention of P4 alone should be understood to refer to $P4_{16}$.

P4 is protocol-independent beyond the physical layer. Any protocol is supported, including completely custom ones. The behavior of the switch is also entirely up to the programmer.

Most switch ASICs that support $P4_{16}$ are based on the portable switch architecture (PSA) [141]. It defines a *pipelined* architecture composed of an *ingress* and an *egress* (*cf.*, figure 3.1). Both are separated in three parts: (i) a parser that extracts data from packet headers according to some rules; (ii) a multi-*stage* match-action pipeline that can route and modify the packets, and

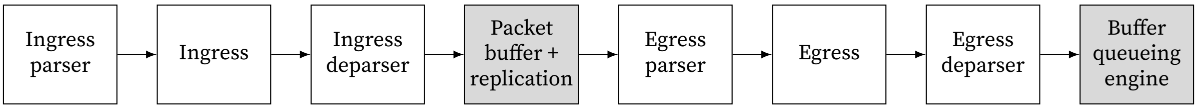

Figure 3.1.: Portable switch architecture pipeline.[2] Blocks represented in white are fully-programmable. Gray blocks are configurable.

[1] Broadcom Trident 4 switches are an exception and use network programming language (NPL) [30].

[2] Inspired by figure 1 in the PSA specification [141].

(iii) a deparser that rearranges internal metadata to a stream of bits. In between the ingress and egress sits a buffer and the replication engine, which means that decisions that change where packets are headed *must* happen in the ingress, and operations on packet replicas *must* happen in the egress.

P4 does not define any way that can create loops to ensure that a program always terminates after a bounded amount of time. However, PSA defines several paths that a packet can take, including *recirculation*, which consists in sending a processed packet back to the beginning of the pipeline for a second round of processing.

3.2. Intel Tofino

Intel Tofino[3] is the first switch ASIC that is fully-programmable by end-users that was released on the market [91]. Tofino implements a proprietary architecture that is close to PSA: Tofino Native Architecture (TNA) [75, 93].

TNA supports stateful operations directly in the data plane through *registers*. It is possible to store a value in a register when a packet matches a set of conditions, and read it back when a later packet matches another set of conditions. Registers are very flexible as they embed an arithmetic–logic unit (ALU) that can perform computations when storing and when reading back its elements.

There are now three versions of Tofino. The original Tofino supports up to 65 Ethernet ports at 100 Gbit/s (6.5 Tbit/s in total). Tofino 2 increases that to 32 ports at 400 Gbit/s (12.8 Tbit/s in total). In 2022, Intel announced the Tofino 3 chip with up to 64 ports at 400 Gbit/s (25.6 Tbit/s in total). These figures highlight that there are no trade-offs in terms of throughput when using a programmable switch instead of a fixed-function one.

[3] Initially known as *Barefoot Tofino* before Intel bought Barefoot Networks, Inc.

Part II.

Native-speed secure data processing

Chapter 4.

Evaluating the cost of computing in trusted execution environments

4.1. Introduction

Thanks to its wide-scale availability on commodity hardware, Intel Software Guard Extensions (SGX) has enjoyed an equally wide adoption as a reference trusted execution environment (TEE), both in academia and in the industry. Its main competitor on the x86 architecture, AMD Secure Encrypted Virtualization (SEV), is rather less sought-after, in part due to being restricted to expensive server-grade processors.

A considerable amount of research papers have explored numerous aspects of Intel SGX, especially in terms of its potential uses. However, despite the existing literature, no extensive experimental study on the impact of these hardware-assisted memory protection mechanisms for memory-bound applications and systems were done until ours. We therefore fill this gap by contributing a detailed performance evaluation study.

First, we fill another gap by contributing a tool to create artificial load within SGX enclaves: **STRESS-SGX**. Thanks to its design, it also doubles as a benchmarking tool.

Using STRESS-SGX and other pieces of benchmarking software, we then measure the performance of Intel SGX and AMD SEV. Noteworthily, we measure the tremendous loss in terms of computing throughput within SGX enclaves when the Spectre security vulnerability [107] is mitigated, an aspect that we came across by accident. In parallel to the performance comparison, we compare the extent of the protection mechanisms of these TEEs.

4.2. STRESS-SGX: a tool to stress-test Intel SGX enclaves

Validating the performance of code running inside an SGX enclave is challenging given the specificity of the technology. It becomes even harder to study the problem under varying conditions, such as different hardware revisions or workloads. In order to simplify the design of future experiments, we introduce the STRESS-SGX tool.

4.2.1. Motivation

We identify three use-cases that can benefit from such tool. First, there exist several real-world traces readily available for research, such as the ones released by Google Borg [182] or Microsoft Azure [39]. These traces allow researchers to replay loads that were observed in an

Table 4.1.: List of stressors supported by Stress-SGX. “X” indicates a stressor available in Stress-ng that is not SGX-compatible. `f`=float, `d`=double, `ld`=longdouble.

ackermann	bitops	callfunc	complex[f,d,ld]	correlate	crc16	decimal[32,64,128]	dither	djb2a	double	euler	explog	factorial	fft	fibonacci	float	fnv1a	gamma	gcd	gray	hamming	hanoi	hyperbolic	idct	int8	int16	int32[f,d,ld]	int64[f,d,ld]	int128[f,d,ld]	jenkin	jmp	ln2	longdouble	loop	matrixprod	nsqrt	ocall	omega	parity	phi	pi	pjw	prime	psi	queens	rand	rand48	rgb	sdbm	sieve	stats	sqrt	trig	union	zeta
✓	✓	✓	X	✓	✓	X	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	X	✓	✓	✓	✓	✓	X	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	X	✓	✓	X	✓	✓	✓	✓	✓

actual cluster of machines. Due to anonymization, hardware characteristics of the original machines remain undisclosed. There has not been any SGX-specific traces released so far. Therefore, available traces do not have the same properties as typical SGX workloads. The nature of jobs deployed on these clusters is also confidential. Stress-SGX can directly map behaviors described in such traces onto similar, SGX-specific ones.

A second motivating scenario stems from the availability of a range of SGX-enabled central processing units (CPUs) on the market, each with their own characteristics and associated performance. Apart from the expected differences due to hardware revision, cache size, available instruction sets or frequency, one still needs to hand-craft micro-benchmarks in order to evaluate SGX-specific performance. Stress-SGX facilitates this process by providing the same interface as Stress-ng [106] to inject load on a CPU under controlled conditions.

A third motivating scenario directly derives from the necessity to measure the electric energy consumption of code executing inside enclaves, by means of software or hardware power meters. Stress-SGX offers a common code-base for both SGX and native contexts, making it easier to isolate the energy requirements of SGX-specific workloads.

4.2.2. Implementation

Stress-SGX is implemented as a fork of Stress-ng (version 0.09.10). It directly reuses the same compilation and runtime foundations. We concentrate on porting CPU and memory stressing methods to run in an enclave in a way that makes both native and SGX versions comparable from a performance standpoint. From a user perspective, SGX-enabled *stressors* are selected by specifying command-line options starting with `--sgx`.[1] All features offered by the CPU and memory stressors shipped with Stress-ng are supported by Stress-SGX, with the exception of partial load. Specifying a fixed load percentage is not possible (*i.e.*, it is locked to 100 %), because this feature is based on precise timing, which is not available in SGX 1 enclaves. Our Stress-SGX prototype supports 54 enclave-enabled stress methods out of the 68 shipped with Stress-ng. The exhaustive list is presented in table 4.1. We note that support for each stressor depends on the availability of its required functionalities within the enclave, as well as their support by the software development kit (SDK).

[1] *Stressors* are specific pieces of executable code that exercise a given low-level operation.

Porting stressors to run inside an enclave

The Intel SGX SDK is designed in a way that allows existing code to be ported to run in an enclave with reasonable engineering efforts [84]. We take advantage of this fact to port the CPU and memory stressors of STRESS-NG in our own STRESS-SGX fork. We detail the particular complications that we encountered throughout this porting effort, and the specific adjustments applied to solve them.

After creating an enclave using the template given by the SDK, we copy the source code of the relevant stressors in the aforementioned template. STRESS-NG defines several macros in its global header file. It is not possible to include the code *verbatim* as it depends on numerous system-specific features. Our solution is to define the needed symbols on a case-by-case basis.

The next obstacle is the need to split the code in *trusted* and *untrusted* parts. We decompose the code in a way that limits the number of enclave transitions (*i.e.*, entering and exiting the enclave) required to run a stressor. Transitions are costly (*cf.*, section 4.3.4); it is therefore crucial that none happen while stressing is in progress to ensure a consistent behavior. Aside from transition-limited stressors, we supplement STRESS-SGX with a stressor called `enclavetransitions` that *only* performs such enclave transitions to allow researchers to study their impact.

The user can gracefully abort the execution of STRESS-SGX using standard Linux signals such as `SIGINT`. STRESS-NG includes a mechanism to catch the majority of signals and react accordingly for its built-in stressors. We leverage the fact that SGX enclaves can access the *untrusted* memory of their enclosing process to pass a pointer to the `g_keep_stressing_flag` variable to the enclave. This variable is later used to indicate when to stop the execution of stressors. Code running inside the enclave periodically polls the flag, and stops the execution if asked by the user. The same flag is also used to make a stressor run for a given duration. Timekeeping is done outside the enclave, with the indication to stop the execution notified by changing the value of the variable.

Ensuring byte-per-byte equivalence of native and SGX code

During the initial testing phase of STRESS-SGX, we observed vast differences in performance for the same stressor executed in native and enclave modes. As a matter of fact, while the source code was identical in both instances, the resulting compiled binaries slightly differed. Conveniently, an enclave compiled using the official SGX SDK will produce a static library. We leverage this aspect to guarantee that both native and SGX versions of a stressor execute a perfectly identical binary by dynamically linking this shared object. Choosing this optional approach limits STRESS-SGX to stressors that are available in enclave mode.

4.3. Performance of Intel SGX

In this section, we want to evaluate what kind of performance we can obtain when running in SGX enclaves compared to native mode. Throughout the experiments, we use a range of SGX-

Table 4.2.: Hardware characteristics of our SGX-compatible test machines.

				Freq. [GHz]	
Category	Model	Processor	Cores	Base	*Turbo*
Server	Supermicro 5019S-M2	Intel Xeon E3-1275 v6	4	3.8	4.2
Desktop	Dell Optiplex 7040	Intel Core i7-6700	4	3.4	4.0
NUC	Intel NUC7i7BNHX1	Intel Core i7-7567U	2	3.5	4.0
Stick	Intel STK2m3W64CC	Intel Core m3-6Y30	2	0.9	2.2

compatible machines, from the very-low footprint Intel Compute Stick up to a rack-mounted server. The specifications of each machine are shown in table 4.2. We use the rack-mounted server in all experiments, and other machines only when specified.

4.3.1. Enclave startup time

We start our evaluation by measuring the time it takes to initialize an SGX enclave. Starting an SGX-enabled program takes longer than a traditional one for two main reasons: support service initialization and memory allocation.

The Intel SGX SDK [90] provides the SGX platform software (PSW) that includes the application enclave service manager (AESM). As its name suggests, AESM is a service that eases the process of deploying enclaves, performing common tasks such as attestation and supporting the access to platform services, like obtaining trusted time and monotonic counters. The other main startup overhead is due to the enclave memory allocation, since all of it must be committed at enclave build time in SGX 1, to be measured for attestation purposes [126].

First, we measure how long starting a traditional process takes. We steadily measure less than 1 ms which is, as we will see, negligible compared to starting an SGX process.

We then quantify the time spent starting AESM. This operation is necessary once per machine

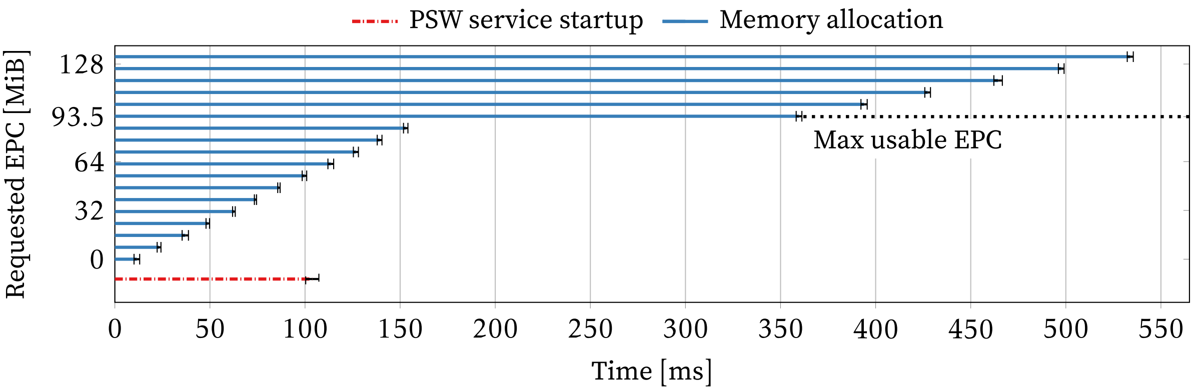

Figure 4.1.: Startup time of SGX processes observed for varying EPC sizes.

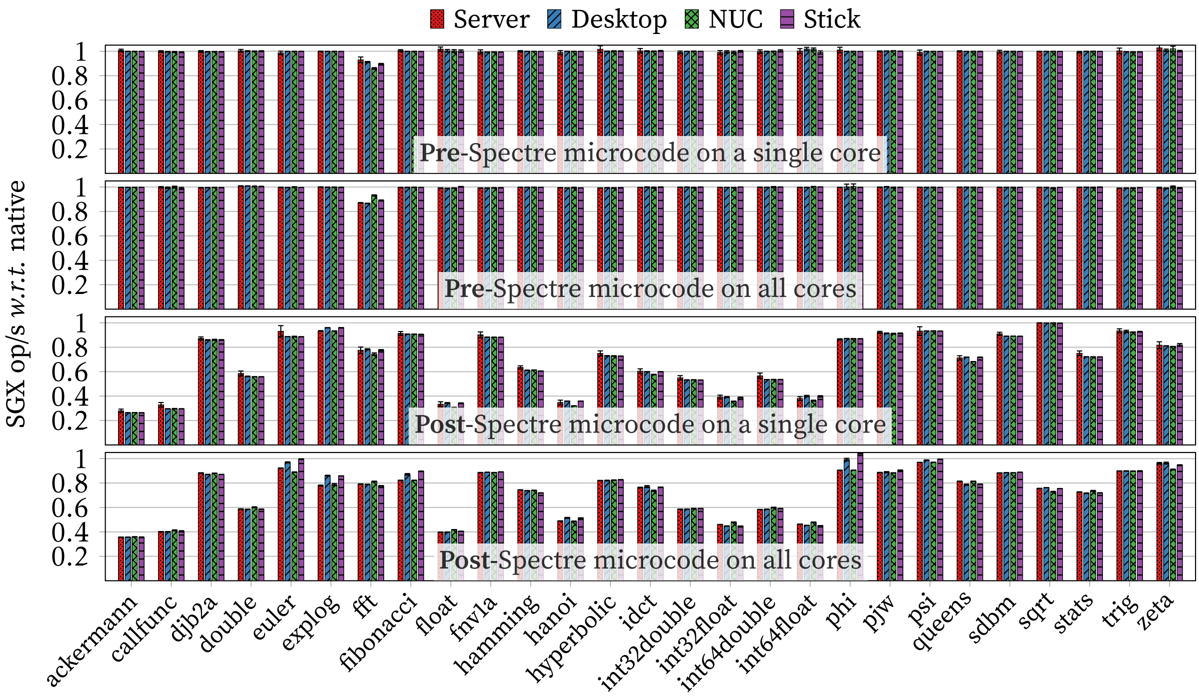

Figure 4.2.: SGX computing throughput compared to native on different types of computers, using pre- and post-Spectre microcodes.

startup, or for each new container in the case of a containerized setup (*cf.*, section 5.4.6). Figure 4.1 shows the average time across 60 runs. Error bars represent the 95 % confidence interval. We can see that starting AESM takes (104 ± 14) ms. That time is fixed regardless of the process that we actually want to execute.

On the other hand, memory allocation time is variable and shows two clear linear trends: before and after reaching the usable enclave page cache (EPC) memory limit. Until this limit, the time increase rate is 1.6 ms/MiB after which it jumps to 4.5 ms/MiB, plus a fixed delay of about 200 ms. Note that these times are just for *allocating* memory, without any actual use of that memory.

4.3.2. Computing throughput

To measure the difference in computing throughput between native and SGX modes, we use the CPU stressors of STRESS-SGX and STRESS-NG. We execute all 54 CPU stressors supported by STRESS-SGX.

Our initial assumption was that the computing throughput in an SGX enclave would be comparable to native, as long as SGX-specific overheads such as enclave transitions or large memory accesses are avoided. In our initial measurements (*cf.*, top two plots in figure 4.2), all stressors

Table 4.3.: Relative speed of memory-bound operations using Intel SGX against native performance. Methods are ordered from sequential (left) to random (right) accesses by increasing memory operation size.

Memory [MiB]	read64	write64	flip	gray	incdec	inc-nybble	walk-0d	walk-1d	rand-set	ror	move-inv	rand-sum	zero-one	galpat-0	galpat-1	swap	modulo-x	prime-gray-0	prime-gray-1	prime-incdec	walk-0a	walk-1a
4	●	●	●	●	●	●	●	◕	●	●	●	●	◕	●	●	●	●	●	◕	●	●	●
8	●	◔	◔	●	●	◕	◕	◕	◑	◕	◑	●	◑	◔	●	◕	◕	●	◑	●	◔	◔
16	◕	◔	◔	●	◕	◕	◑	◕	◔	◔	◔	●	◔	◔	◕	◑	◔	●	◔	●	◔	◔
64	◕	◔	◔	●	◕	◕	◑	◕	◔	◔	◔	◕	◔	◔	◕	◑	◔	●	◔	●	◔	◔
256	○	○	○	◔	○	○	○	○	○	○	○	○	○	○	○	○	○	◑	○	◑	○	○

indeed run at the same speed within an SGX enclave as natively—the only exception being the *fft* stressor for an unknown reason.

Later during the evaluation, we discovered that some of our test machines with Skylake processors exhibited underwhelming SGX performance compared to other similarly-equipped machines. After some investigations and further benchmarks, we found out that the only difference between these machines was their microcode versions (*cf.*, section 2.1.1). We observed that the microcode update issued by Intel on 2018-01-08 was the first version that exhibited the performance degradation. It was released to mitigate the infamous Spectre attack [107].

Under the updated microcode, we observe a significant difference in SGX *versus* native performance on all types of machines. The bottom two plots in figure 4.2 present the results for the 27 tests for which SGX performance is affected. We measure slowdowns up to 3.8× (`ackermann` running on a single core). Given the undisclosed nature of microcode updates, it is difficult to identify the exact root cause of this performance degradation.

4.3.3. Memory accesses

Aside from pure computing throughput, we measure the throughput of memory accesses in SGX enclaves compared to native. Memory accesses in SGX 1 use the EPC, and may need to swap pages to regular memory should its 93.5 MiB capacity be exceeded.

We rely on the virtual memory stressors of STRESS-NG as a baseline. We then use the same stressors, but running in STRESS-SGX, to run the same benchmark in SGX enclaves. We ensure that both SGX-protected and unprotected versions of the stressors execute the exact same binary code, to provide results that can be directly compared against one another. In STRESS-NG, we replace the memory allocation function `mmap` used by the virtual memory stressors

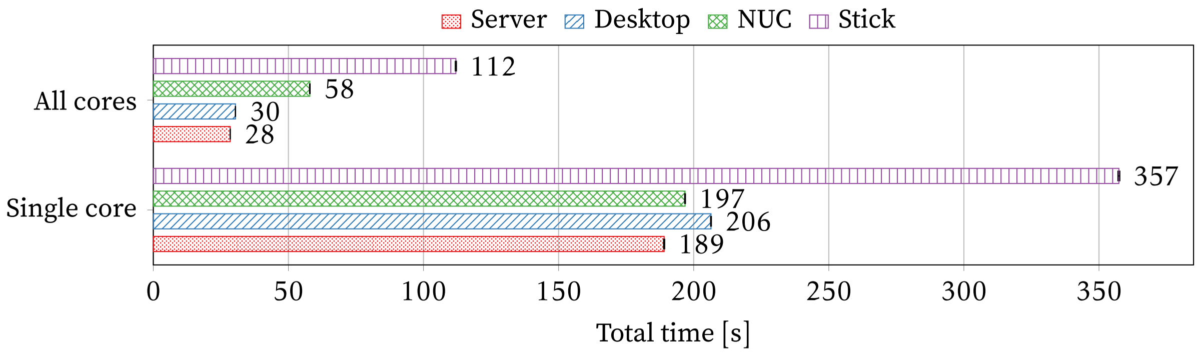

Figure 4.3.: Time needed to perform 100 000 000 enclave transitions.

with `malloc` to obtain a fair comparison between native and SGX (which prohibits `mmap` due to its use of system calls).

Table 4.3 summarizes the results of this benchmark. It can be read in the following way: the colored surface of each disk represents the relative execution speed in an SGX enclave compared to the native speed on the same machine for the same configuration. For example, a disk that is 75 % full (◕) indicates that a stressor ran with protection mechanisms enabled at 0.75× the speed observed in native mode. A full disk (●) indicates that the performance of the associated stressor is not affected by the activation of SGX.

Values are taken from the average of 10 executions, where each method is spawning 4 stressors with an execution limit of 30 s.

The protection mechanisms of SGX 1 only encrypt data leaving the CPU package. Our test machine has 8 MiB of cache at various levels. The performance is hence comparable to native mode when the program operates on 4 MiB of memory. In this case, the data never leaves the die and is therefore processed and stored in clear. The stressors *walk-1d*, *zero-one*, and *prime-gray-1* are the ones that are the most affected by the activation of SGX, executing at 80 % speed compared to native when operating on 4 MiB.

When we increase the size of memory accesses beyond the size of the EPC, we see that the throughput under SGX goes down to less than 5 % of the native performance. Under these conditions, methods such as *modulo-x* were not able to produce any results. However, supplemental tests, during which hyper-threading was enabled and all 8 virtual CPUs used, did return results.

SGX performs better when memory accesses follow a sequential pattern, as observed in the tests *read64, gray, incdec, inc-nybble, walk-d1* and *rand-sum*. Conversely, it is negatively affected by random memory accesses, as exhibited in the results of tests *swap, modulo-x, prime-gray-1, walk-0a* and *walk-1a*.

Table 4.4.: Hardware characteristics of our SEV-compatible test machine.

			Freq. [GHz]	
Model	Processor	Cores	Base	*Turbo*
Supermicro 1023US-TR4	2× AMD EPYC 7281	2×16	2.1	2.7

4.3.4. Enclave mode transitions

The most common performance overhead of an SGX-enabled program is the transition between protected and unprotected modes [11, 144, 29, 140]. When programming for SGX, it is important to keep that aspect in mind.

We use the `enclavetransitions` stressor of STRESS-SGX to quantify the time needed to perform 100 million enclave transitions. Figure 4.3 presents the results on a single core at a time and on all available cores. On a single core, the results show that the frequency of each processor is the determining factor to transition into an SGX enclave faster. The multicore case shows that transitioning between normal and SGX modes is a highly parallelizable process.

4.4. Performance comparison between Intel SGX and AMD SEV

SEV is another TEE for the x86 architecture (*cf.*, section 2.2). In this section, we compare its performance to Intel SGX 1.

An important difference to consider is the amount of memory that can effectively be used by programs running in SGX 1 enclaves compared to SEV virtual machines (VMs). The EPC area used by SGX 1 enclaves is limited to 128 MiB, of which 93.5 MiB are usable in practice by applications. More memory can be used but at the expense of paging to main memory. This limit does not exist for SEV: applications running inside an encrypted VM can use all its allocated memory.

Intel SGX has data-integrity protection mechanisms built-in. AMD SEV, on the other hand, does not provide any integrity protection mechanism [132]. This limitation is addressed by the SEV Secure Nested Paging (SEV-SNP) extension [2].

We execute our benchmarks on server-grade machines (*cf.*, tables 4.2 and 4.4). As our SGX-compatible server is less powerful than our SEV-compatible one, we deploy para-virtualized VMs that are limited to 4 virtual CPUs and 16 GiB of random access memory (RAM) on the AMD machine. Note that our AMD-based test machine only supports the original iteration of SEV (no support for SEV Encrypted State (SEV-ES) or SEV-SNP).

Table 4.5.: Relative speed of memory-bound operations using AMD SEV against native performance. Counterpart to table 4.3.

Memory [MiB]	read64	write64	flip	gray	incdec	inc-nybble	walk-0d	walk-1d	rand-set	ror	move-inv	rand-sum	zero-one	galpat-0	galpat-1	swap	modulo-x	prime-gray-0	prime-gray-1	prime-incdec	walk-0a	walk-1a
4	●	●	●	●	◕	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	◕	●
8	●	●	●	●	●	●	●	●	◕	◕	●	●	●	◕	●	●	◕	●	●	●	◕	◕
16	●	●	◕	●	◕	●	●	●	◕	◕	●	●	●	◕	●	◕	●	●	●	●	◕	◕
64	●	●	●	●	◕	●	●	●	◕	◕	◕	●	●	◕	●	◕	●	●	●	●	●	◕
256	●	●	●	●	●	●	●	●	◑	◑	◑	●	●	◕	●	◕	◕	●	●	●	◕	◕

4.4.1. Memory-bound operations

We begin by comparing the memory access overhead of AMD SEV to Intel SGX. We repeat the experiment described in section 4.3.3, with adaptations for AMD SEV. The benchmark stills consists in the virtual memory stressors of STRESS-NG. As SEV only applies to virtual machines, our baseline runs in a traditional virtual machine. Subsequently, the benchmark runs with SEV enabled.

Table 4.5 shows the results of this micro-benchmark. The results can be compared to SGX, as shown in table 4.3 on page 20. Values are also taken from the average of 10 executions, where each method is spawning 4 stressors with an execution limit of 30 s. Each disk in the figure represents the relative execution speed in SEV protected mode, compared to native speed on the same machine for the same configuration.

At first sight, SEV appears to have much less overheads than SGX (disks are fuller and greener overall), mostly due to its lack of integrity protection. Similarly to Intel SGX, performance is not affected when the program operates on a small amount of memory (*i.e.*, 4 MiB). The reason is that the protection mechanisms are only used to encrypt data leaving the CPU die, and 4 MiB is smaller than the amount of cache embedded on the CPU. Again, both technologies perform better when memory accesses follow a sequential pattern (*read64, gray, incdec, inc-nybble, walk-d1, rand-sum*) than for random memory accesses (*swap, modulo-x, walk-0a, walk-1a*).

Unlike SGX, it is possible to operate over larger amounts of memory (*i.e.*, 256 MiB) with AMD SEV enabled without encountering any measurable overhead. Programs with larger trusted computing bases (TCBs) are hence practical with AMD SEV.

4.4.2. Caching effects

In order to show the impact of the various levels of processor caches on SGX and SEV performance, we measure the throughput for varying sizes of memory accesses. We use the

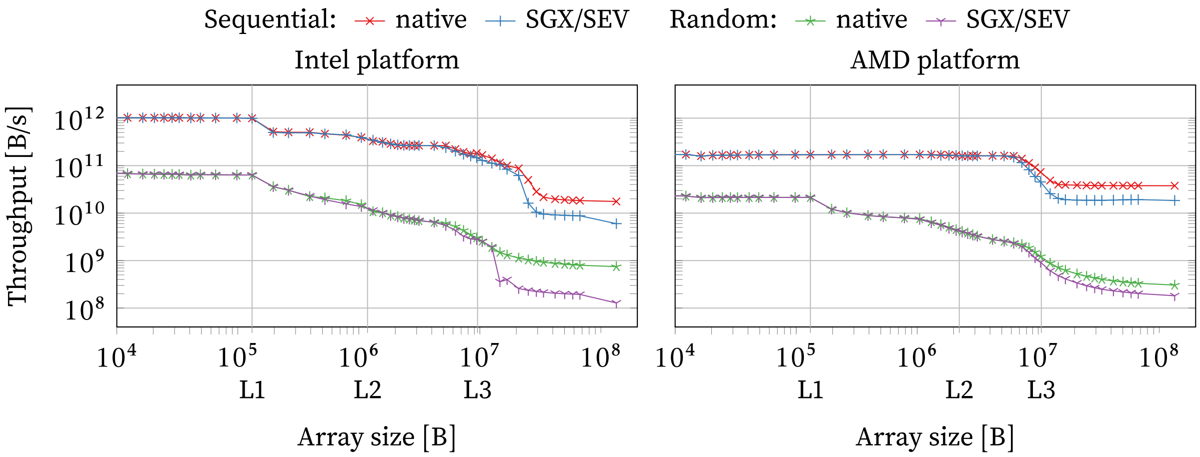

Figure 4.4.: Measured throughput when sequentially and randomly accessing a memory region of varying sizes, natively and in Intel SGX/AMD SEV.

pmbw tool [24] to conduct this experiment. We port it to run in SGX enclaves with Graphene-SGX [170]. Likewise, we use Graphene [169] to run the native case on the Intel platform so as to provide a comparable baseline. On the AMD platform, we run *pmbw* in a VM, with and without SEV enabled. We pin the 4 virtual CPU cores to physical CPU cores within a single core complex (CCX) [160].

Figure 4.4 shows the observed throughput averaged over 10 runs when reading through a fixed amount of memory. The results are presented in the form of a *log-log* plot. We highlight the cumulative sizes of the L1, L2 and L3 caches.[2] We see that, within the first two levels of cache, the performance of both AMD SEV and Intel SGX is strictly equivalent to native performance. SEV shows some overhead within the L3 cache for sequential and random accesses, while SGX keeps in line with native performance at that level. However, larger memory accesses are differently affected. In the case of SEV, only a small overhead can be observed on sequential and random accesses. On the other hand, the paging mechanism of SGX greatly affects the throughput we measure, in particular in the case of random memory accesses.

We observe a couple of surprising abnormalities when looking at our results. First, on the Intel platform, the drop in performance does not happen at the cumulative size of caches, but beyond. Our hypothesis is that this behavior is caused by Intel *smart cache* technology [167]. Second, on the AMD platform, the throughput within L3 cache is decreasing at a much faster rate than on the Intel platform. We observe the same behavior in native and SEV-protected modes. We believe that the performance decrease is due to the virtualization process of the VM.

[2] In the case of the AMD machine, the values represent a single CCX.

4.5. Summary

The performance experiments we presented in this chapter show that the original version of AMD SEV has very little performance impact, especially when compared to Intel SGX 1. However, its much weaker security guarantees (*cf.*, section 2.2 and tables 2.1 and 2.2) make it hard to find a compelling ad-hoc use-case for this piece of technology. Its advantage lies in the fact that no modifications need to be done to existing programs to benefit from the security guarantees it offers.

Regardless, we show that Intel SGX 1 can also have no performance impact in some cases, but can impose tremendous penalties depending on one's usage. Our study highlights that developers need to be continually aware of the performance shortcomings of SGX 1 to make the best use of it. Its extensive security promises hence make it better suited to ad-hoc TEE-enhanced use-cases.

Chapter 5.

Orchestration of TEE-secured programs on heterogeneous clusters

5.1. Introduction

A recent trend is to package and deploy applications in the form of containers. They offer reproducible execution environments, light lifecycle management and closer-to-metal performance than hypervisor-based solutions.

Developers can rely on turn-key container *images* available from public and private container *registries*. Most cloud operators directly support container deployments as part of their commercial offering (*e.g.*, Google Container Engine [62] or Amazon EC2 Container Service [6]). Similarly, it is possible to set up a container cluster on private premises, leveraging popular [43] container *orchestrators* such as Kubernetes [111] or Docker Swarm mode [45].

Without special care, containers are exposed to critical security threats. For instance, the cloud infrastructure could be compromised by malicious actors or software bugs. This is especially true when containers are deployed on a public cloud infrastructure, but it also holds true for the case of a deployment on private premises that may have been corrupted by malicious actors. Hence, users who deploy services are left with no other choice than to trust the infrastructure provider and the complete software stack (including the underlying operating system, kernel libraries, *etc.*). Similarly, they must face the risk that a compromised component can lead to severe data leakage [53, 164].

A solution is to use Intel SGX enclaves to shield critical services from malicious actors. The availability of SGX allows to distrust the cloud operator and rely instead on hardware protection mechanisms, hence drastically reducing the TCB. Cloud providers are starting to offer SGX-enabled infrastructure as a service (IaaS) solutions to end-users. One example is Microsoft Azure Confidential [154].

Deploying and orchestrating containers on a heterogeneous cluster, with a mix of machines with and without SGX capabilities, presents its own set of specific challenges. The containers that require SGX will contend on the availability of dedicated memory, due to its constrained nature. Therefore, there needs to be a monitoring infrastructure that feeds the scheduler with resource usage metrics of SGX memory requests to schedule the containers accordingly. Unfortunately, none of the existing container orchestrators offer native support to provide runtime insights about the resources used by SGX containers.

In the proposed context, our contributions are the following. We propose an SGX-aware architecture for orchestrating containers. Fitting in this architecture, we offer **SK8S**,[1] an open-source vertical implementation of the required system support [172]. SK8S includes modifications to the Linux module for SGX as well as a Kubernetes *device plugin*. Further, we show that it will be possible to drastically reduce the waiting time of the submitted jobs by exploiting SGX 2, as it offers better control over the size of the dedicated memory. Finally, we demonstrate that our design and implementation are sound with a detailed evaluation using the Google Borg traces [182, 150].

5.2. Related work

Several research efforts have tackled the problem of scheduling jobs over a cluster of heterogeneous machines. To protect sensitive hypervisor scheduling decisions, *Scotch* [118] conveys information gathered in system management mode (SMM) to enclaves. They implement a prototype on top of the Xen hypervisor, adding about 14 % of overhead for each context switch and interrupt. Although they provide results within 2 % of the ground truth in such scenarios, there is no guarantee that measurements coming from SMM are not tampered with during control switch to enclave entry points. Their focus is on the protection of probing data and preventing improper resource usage. Instead, we deal with system support for scheduling SGX jobs based on their main contentious resource: EPC memory pages.

Similar to our work, *ConVGPU* [100] provides solutions for scheduling jobs based on memory contention in container-based virtualized environments. Specifically, they provide a mechanism that shares graphics processing unit (GPU) memory among multiple containers. Just as the EPC, GPU memory is limited. However, it is not possible to swap out memory once it is full, an event that usually leads to more severe issues than just performance degradation. To avoid that, ConVGPU intercepts and keeps track of memory allocation calls to the CUDA application programming interface (API) by providing an alternative shared library to applications running within containers. Whenever a request cannot be granted, it preempts the application by postponing the return call until there are available resources. They evaluate the system using four strategies for the selection of which waiting application should be served first, and show low overall application running time overheads. Essentially, they act reactively to potential memory contention issues, after container deployment, whereas we take scheduling decisions before, based on self-declared memory needs, and after deployment, based on probed data. Besides, they only take into account intra-node resource management, and leave distributed processing by integration with Docker Swarm for future work.

MixHeter [187] also deals with scheduling in heterogeneous environments. Since different sorts of applications benefit from distinct hardware capabilities (*e.g.*, GPUs for graph computing, RAM for sorting jobs, *etc.*), distributed system schedulers that deal with mixed workloads and take decisions disregarding this aspect may face poor performance. To that end, *MixHeter* proposes a scheduler based on *or-constraints*. The different resource requests are translated

[1] Pronounced “skates”.

into algebraic expressions to be satisfied. If preferred resources are busy, non-preferred, but still compatible ones are used instead, which can maximize the overall performance. They evaluate the system on a popular scheduler for distributed systems processing frameworks and show performance improvements up to 60 %. We only consider SGX and non-SGX jobs as characterizing features for orchestration decisions. Our system would therefore benefit from such a scheme considering a broader range of hardware capabilities, assuming applications would support alternative TEEs (*e.g.*, AMD SEV, ARM TrustZone, and even trusted platform modules (TPMs) [10]) in the absence of Intel SGX.

Checkpointing and migration of running processes closely relates to scheduling strategies. In this direction, Gu *et al.* [67] tackle these issues for SGX enclaves. The challenge lies in securely creating, transmitting and restoring an enclave checkpoint while preserving all security guarantees, while not introducing new attack vectors. Checkpointing a running SGX process, however, already imposes some obstacles, since part of the enclave metadata is not even accessible by the enclave itself. Moreover, for consistency reasons, one must ensure that all application threads do not continuously mutate the state of the job being migrated. The authors deal with the first issue by inferring the value of such unreadable metadata. Then, they issue replay operations that lead to an identical state. However, their approach relies on the cooperation of the untrusted operating system, and therefore checks afterwards if it has behaved accordingly. The problem of achieving a *quiescent* point, when all threads are guaranteed to not modify the process state, is done by synchronization variables kept inside the enclave, and by intercepting its entry and exit points. By doing this, they force all threads to reach either a dormant or a spinning state that will only be undone after restoring the enclave at the target node. After successfully creating the checkpoint, they still have to provide means to ensure that it cannot be restored more than once (fork attack) nor that an old one can possibly be recovered (rollback attack). That is solved by means of a migration key transmitted through secure channels built by leveraging SGX attestation and by a self-destroy approach, which prevents the enclave from being resumed after it was checkpointed. Overall, their system shows a negligible performance overhead. Such mechanism could eventually be integrated into our system, towards a globally-optimized EPC utilization through the migration of enclaves. However, we stress that the problem of SGX enclave migration (online or offline) is considered orthogonal to ours.

5.3. Design

5.3.1. Trust model

In order to accurately design our orchestrator, we need to decide upon its trust model. We assume that our SGX-enabled orchestrator is deployed on the premises of a given public cloud provider. Providers show a honest-but-curious behavior. They are interested in offering an efficient service to customers, mostly for selfish economic reasons (*e.g.*, providers want to maximize the number of executed jobs per unit of time, but they will not deliberately disrupt them). However, providers do not trust their customers, especially not the resource usage

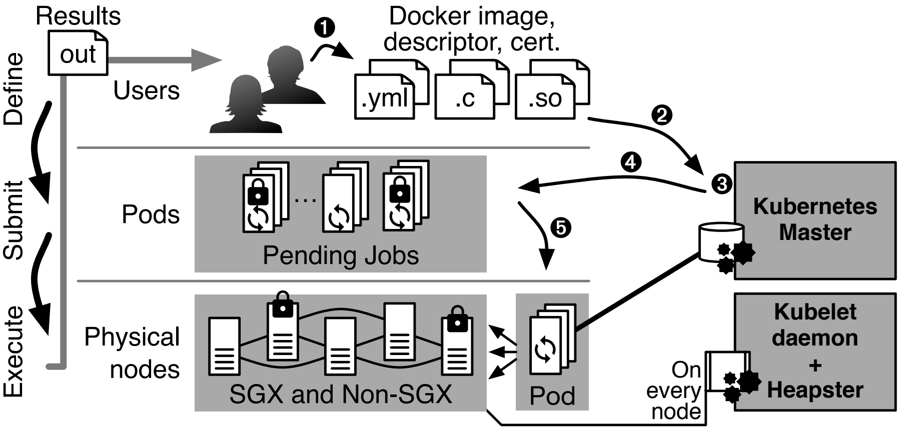

Figure 5.1.: Architecture and workflow of the SGX-aware orchestrator.

declarations they specify at deployment time. Nevertheless, providers trust their own infrastructure, namely the operating system running on the cluster nodes, the SGX driver, as well as the orchestrator itself.

Customers rely on the infrastructure offered by cloud providers, but do not trust it. Notably, they trust Intel SGX to prevent providers from inspecting the contents or tampering with jobs deployed on their infrastructure. In turn, customers might try to allocate more resources than what they requested, at the cost of providers or other tenants.

5.3.2. Architecture

The architecture of our system is depicted in figure 5.1. The complete system implements a container orchestrator that can efficiently schedule SGX-enabled jobs, as well as regular jobs, on a heterogeneous cluster. We assume that SGX-enabled jobs execute entirely in enclaves, minus a part responsible for bootstrapping the SGX enclaves. These assumptions match those of state-of-the-art SGX systems, *e.g.*, SCONE [11] or Graphene-SGX [170].

One important aspect of the system is its ability to guarantee that jobs submitted to a given host always fit within its EPC memory limits. This is of particular relevance to avoid major performance penalty (*cf.*, section 4.3.3). To achieve this goal, we use a monitoring layer, implemented by means of daemons executing on each node of the cluster. They accurately measure individual EPC utilization of the jobs submitted to our system. These measures are periodically sent to a central node, in charge of storing and analyzing them. We rely on a time-series database for this task, as they have proved to be an efficient tool to implement queries over moving sliding windows [142, 51, 79], such as the ones we use in our scheduling policy.

Users submit their jobs by specifying the name of the container image (*cf.*, figure 5.1-❶). They need to indicate the amount of EPC memory required by their jobs at this time.[2] Apart from these parameters, clients rely on the regular APIs of the orchestrator. The image is initially pulled from a public or private container registry. Afterwards, it is submitted into the scheduler's queue as a pending job ❷.

5.3.3. Scheduling algorithm

Our scheduling algorithm works as follows. First, the container orchestrator fetches the list of pending jobs currently in the queue ❸. It takes their memory allocation requests into account, both in terms of standard memory and EPC. At the same time, it fetches accurate and up-to-date metrics about memory usage across all nodes in the cluster. This is done by executing a sliding-window query over the time-series database. The scheduler then combines the two kinds of data to filter out job-node combinations that cannot be satisfied, either due to hardware compatibility (*i.e.*, SGX-enabled job on a non-SGX node), or if the job requests would saturate a node. The next step depends on the concrete container placement policy configured for the cluster. In particular, we enable support for SGX measures in two well-known placement policies.

When *binpack* is in use, the scheduler always tries to fit as many jobs as possible on the same node. As soon as its resources become insufficient, the scheduler advances to the next node in the pool. The order of the nodes stays consistent by always sorting them in the same way. In the case of a standard job, we sort SGX-enabled nodes at the end of this list, to preserve their resources for SGX-enabled jobs.

Conversely, the main goal of the *spread* strategy is to even out the load across all nodes. It works by choosing job-node combinations that yield the smallest standard deviation of load across the nodes. Like *binpack*, it only resorts to SGX-enabled nodes for non-SGX jobs when no other choice is possible to execute the job. After the policy selection is made, the scheduler communicates the computed job-node assignments to the orchestrator ❹. It then handles the actual deployment of jobs towards the various nodes ❺.

There may be jobs that cannot be fitted in the cluster at the time of their submission. The orchestrator keeps a persistent queue of pending jobs ❸; the scheduler periodically checks for the possibility to schedule some of them, applying a first come first serve (FCFS) priority.

5.4. Implementation

In this section, we provide insights on the implementation of the components of our architecture.

[2] As the allocation of EPC memory has to be done at program initialization for SGX 1 enclaves, this value is written in the binary and could be extracted. We rely on the user's specification for convenience reasons.

As several mainstream container orchestrators exist today, we decide to build our implementation on top of the Kubernetes container orchestrator [111].[3] The entire source code of our implementation is released as free software [172]. The components that we add to the architecture of Kubernetes interact with it using its public API. This approach facilitates integration into future versions of Kubernetes.

5.4.1. Kubernetes device plugin

The first component is a *device plugin* [110] that allows to mark a Kubernetes node as able to execute SGX instructions. Those kind of plugins can be used directly by the *Kubelet* node agent since Kubernetes 1.8. The design behind device plugins was influenced by the developers' need to access GPUs [57]. Its intent is to expose system devices (*i.e.*, those available in the `/dev` pseudo-filesystem) directly within Kubernetes. Opportunely, applications implemented using the Intel SGX SDK for Linux are given access to SGX by means of a pseudo-file in `/dev`.

Communication between Kubelet and the device plugin leverages Google remote procedure calls (gRPC) [66]. Our device plugin checks for the availability of the Intel SGX kernel module on each node and reports it to Kubelet. Kubelet notifies the control plane about the availability of an "SGX" resource on that node.

The philosophy behind device plugins is to register one *resource item* (*e.g.*, one GPU card, one field-programmable gate array (FPGA) board, *etc.*) per physical device. In the case of SGX applications, there is only one pseudo-file registered per processor. However, multiple enclaves can be executed at the same time, sharing the EPC. Exposing only one resource item would have utterly limited the usefulness of our contribution, as only one SGX-enabled set of containers (or *pod* in the Kubernetes terminology) could be scheduled on a physical host at any given time. We solve this problem by exposing each EPC page as a separate resource item. By exposing the EPC as multiple independent pseudo devices, several pods can be deployed and share a single node, thus supporting the execution of several SGX applications at once. Despite the great amount of resources created with this scheme, we did not notice any perceptible negative influence on performance.

Although SGX can over-commit its protected memory via paging, doing so leads to severe performance drops. Therefore, we deliberately prevent over-commitment of the EPC, in order to preserve predictable performance for all pods deployed in the cluster.

Thanks to the device plugin, Kubernetes knows that it needs to mount the SGX pseudo-device file inside each pod that requested at least one share of EPC. Therefore, end-users must declare that their SGX-enabled pods use some amount of the "SGX" resource. In the case of Kubernetes, this is done by filling in the resource requests and limits fields of the pod specification. Resource requests are used by the scheduler to dispatch SGX-enabled pods towards a suitable node. Limits are transmitted to our modified SGX kernel module for strict enforcement.

[3] An initial attempt was based on Docker Classic Swarm. However, its lack of hooks to extend its architecture and poor developer documentation convinced us to look for a different solution.

5.4.2. SGX-aware scheduler

Once nodes have been configured to use our device plugin, they are ready to accept pods using Intel SGX instructions. However, Kubernetes' own scheduler only relies on values communicated in the resource requirements of each pod. Given the restrained capacity of the EPC, it is imperative to maximize its utilization factor. To do so, the scheduler must consider the actual usage metrics of the cluster, collected at runtime. Our scheduler can decide on scheduling actions based on measured memory usage—for the EPC as well as regular RAM. Metrics regarding regular memory are collected by *Heapster* [101], while SGX-related metrics are gathered using our custom probes.

We implement the previously-described *binpack* and *spread* scheduling strategies in a non-preemptive manner, following the same scheme as the default scheduler of Kubernetes. The scheduler itself is packaged as a *pod*. This allows us to execute it with the same privileges as the default scheduler. It also provides us with seamless migrations and crash monitoring features, as for any pod.

Kubernetes can cope with multiple schedulers concurrently operating over the same cluster. It is therefore possible to deploy our scheduler with both of its strategies in parallel to the default, non SGX-aware one. Comparative benchmarking is thus made easier, as each pod deployed to the cluster can specify which scheduler it requires. We assume that, in production deployments, only one variant of our SGX-aware scheduler will be deployed as default to prevent conflicts.

5.4.3. SGX metrics probe

The proposed scheduling algorithm relies on metrics directly fetched from nodes of the cluster. Kubernetes natively supports Heapster [101], a lightweight monitoring framework for containers. We configure Heapster to collect such metrics on each node and subsequently store them into an InfluxDB [82] time-series database. Moreover, we have implemented an SGX-aware metrics probe to gather EPC usage metrics from our modified Intel SGX kernel module. These metrics are pushed into the same InfluxDB database used by Heapster. This allows our scheduler to use equivalent InfluxQL queries [83] against the database for SGX- and non SGX-related metrics.

The probe is deployed on all SGX-enabled nodes using the *DaemonSet* component [112]. The distinction between standard and SGX-enabled cluster nodes is made by checking for the EPC size advertised to Kubernetes by our device plugin. Finally, we leverage Kubernetes itself to automatically handle the deployment of new probes when adding physical nodes to the cluster, as well as their management in case of crashes.

5.4.4. Enforcing limits on EPC usage

SGX supports over-commitment of its primary cache, the EPC. However, this feature comes with an important performance penalty for user applications. It is imperative for a cloud provider to make sure that multiple containers share the EPC in a respectful manner. In Kubernetes, users specify the limits for each type of resource that their pods use. These values are later used for several purposes, *e.g.*, accounting and billing the reserved resources. It is in the interest of the infrastructure provider to make sure that co-hosted containers do not contend on the same resource. A user with malicious intents could advertise lower amounts of resources than what their pods actually use. For this reason, it is crucial to enforce the limits advertised in the specification of each pod. In our particular context, we focus on limits related to EPC usage.

We implement proper limits enforcement by modifying two existing components of our architecture: (i) the Linux version of the SGX driver provided by Intel, and (ii) Kubelet, the daemon running on each node of a Kubernetes cluster. The SGX kernel module will deny the initialization of any enclave that exceeds the share of pages advertised by its enclosing pod.

Linux cgroups

The proper way to implement resource limits in Linux is by adding a new *cgroup* controller to the kernel [128]. This represents a substantial engineering and implementation effort, affecting several layers of our architecture. Modifications would be required in Kubernetes, Docker (which Kubernetes uses as container runtime) and the Linux kernel itself.

We considered a simpler, more straightforward alternative. Namely, we use the *cgroup* path as a pod identifier. The rationale behind the choice of this identifier is as follows: (i) it is readily available in Kubelet and in the kernel; (ii) all containers in a pod share the same *cgroup* path, but distinct pods use different ones, and (iii) the path is available before containers actually start, so the kernel module knows the limits applicable to a particular enclave on its initialization.

In order to communicate limits from Kubernetes to the SGX kernel module, we added 16 lines of Go code and 22 lines of C code to Kubelet, using *cgo* [60]. These additions communicate a new *cgroup path–EPC pages limit* pair each time a pod is created.

5.4.5. Modified Intel SGX kernel module

We modify the Linux kernel module [85] provided by Intel. Our modifications add the following features: (i) gathering of usage data to improve scheduling decisions, and (ii) enforcement of resource usage limits. We offer access to the total number of EPC pages, as well as their current usage status by way of module parameters. They are accessible using the usual Linux filesystem interface, below the `/sys/module/isgx/parameters` path. Values can be retrieved through two pseudo-files:

`sgx_nr_total_epc_pages`: total amount of pages on the system.

`sgx_nr_free_pages`: amount of pages not allocated to a particular enclave.

Additionally, per-process EPC usage can be queried. To do so, we create a new input/output queryable using the `ioctl` function [96] available in Linux. This control reports the number of occupied EPC pages given to a single process, described by its process identifier (PID). This metric is helpful to identify processes that should be preempted and possibly migrated, a feature especially useful in scenarios of high contention.

We create a second input/output queryable to communicate resource usage limits. Each pod deployed in our cluster needs to advertise the number of EPC pages it plans to use. When an SGX-enabled pod is created, Kubelet calls the `ioctl` function to communicate its EPC usage limits to the kernel module. The module makes sure that limits can only be set once for each pod, therefore preventing a privileged container from resetting them. With SGX 1, enclaves must allocate all chunks of protected memory that they plan to use at initialization time. We add a couple lines of code to the function `__sgx_encl_init` that calls our own function that checks whether to allow or deny a given enclave initialization. Internally, it compares the number of pages owned by the enclave to the limits advertised by its enclosing pod.

The implementation of these features consists in 115 lines of C code on top of the Intel SGX Linux kernel module [174].

5.4.6. Base image to use Intel SGX in containers

When an application wants to use the processor features offered by Intel SGX, it has to operate in enclave mode. With SGX 1, the program has to pre-allocate all the enclave memory that it plans to use. The particular x86 instructions that can reserve enclave memory can only be executed by privileged code running in *ring 0* [40]. Under the GNU/Linux operating system, only the kernel and its modules are allowed to call these instructions. In a containerized context, the kernel and, by extension, its modules are shared across all containers. In the case of running SGX-enabled applications in containers, this implies that it is required to set-up a communication channel between the container and the SGX module. In Docker, this can be done by mounting the device pseudo-file exposed by the host kernel directly into the container. While it is possible to create SGX-enabled applications that directly interface with the kernel module, the SDK provides an easier path to SGX application development. Programs that use the SDK rely on the PSW [90]. Hence, we created a Docker image that allows SGX-enabled applications developed using the official SDK to be executed seamlessly within Docker containers. The source of the image is publicly available from GitHub [171]. Our image supports versions 1.9 to 2.5 of the SGX SDK. Starting from version 2.6 of the SGX SDK, Intel provides an official Docker image [86].

5.5. Evaluation

In this Evaluation follows a detailed evaluation of our SGX-aware scheduler. First, we describe our experimental settings. Then, we characterize the synthetic workload that we use as well as the Google Borg trace and the simplifications that adapt it to the scale of our cluster. A comprehensive evaluation of the scheduler itself follows, including measurements of the effectiveness of strictly enforcing resource usage limits. We end our evaluation by investigating how our work is affected by the release of SGX 2.

5.5.1. Evaluation settings

For this evaluation, we use a cluster of 5 machines. The first 3 machines are Dell PowerEdge R330 servers, each equipped with an Intel Xeon E3-1270 v6 CPU and 64 GiB of RAM. At the time of experimentation, these machines were not compatible with Intel SGX. One of these machines acts as the control plane for Kubernetes, while the remaining are regular Kubernetes nodes.

The two remaining nodes are SGX-enabled machines, also acting as nodes in the Kubernetes cluster. These machines feature an Intel i7-6700 CPU and 8 GiB of RAM. SGX is statically configured to reserve 128 MiB of RAM for the EPC. The machines are connected to a 1 Gbit/s switched network. We use Kubernetes (v1.8), installed on top of Ubuntu 16.04. We enable the Kubernetes *device plugin* alpha feature on all the machines.

5.5.2. The Google Borg trace

Our evaluation uses the Google Borg trace [182, 150]. The trace was recorded in 2011 on a Google cluster of about 12 500 machines. The nature of the jobs in the trace is undisclosed. We are not aware of any publicly available trace that would contain SGX-enabled jobs. Therefore, we arbitrarily designate a subset of trace jobs as SGX-enabled. In the following experiments, we insert various percentages of SGX jobs in the system: from 0 % of SGX jobs (only standard jobs), and then increasing by 25 % steps, until 100 % (only SGX jobs).

The trace reports several metrics that were measured in Google's cluster. We extract the following metrics out of it: (i) submission time; (ii) duration; (iii) assigned memory, and (iv) maximal memory usage. The submission time and duration allow us to replicate the arrival pattern of jobs in our cluster. We use the *assigned memory* field as the value advertised to Kubernetes when submitting the job to the system. However, the job will actually allocate the amount given in the *maximal memory usage* field. We believe this creates real-world-like behavior with respect to the memory consumption advertised on creation compared to the memory that is actually used.

The trace specifies the memory usage of each job as a percentage of the largest memory capacity in Google's cluster—without actually reporting the absolute value. Figure 5.2 shows the amounts of memory allocations recorded in the trace. In our experiments, we set the memory usage of SGX-enabled jobs by multiplying the memory usage factor obtained from the trace to the total usable size of the EPC (93.5 MiB in our case). As for standard jobs, we compute their memory usage by multiplying them to 32 GiB. The rationale behind this choice is that it is the power-of-2 closest to the average of the total memory installed in our test machines. Moreover, we think that it yields amounts that match real-world values.

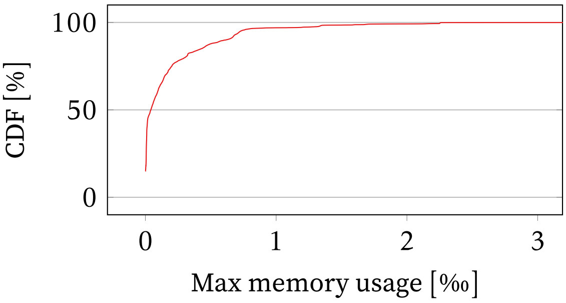

Figure 5.2.: Borg trace: distribution of maximal memory usage.

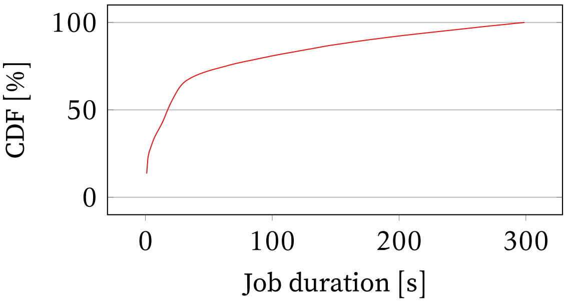

Figure 5.3.: Borg trace: distribution of job duration.

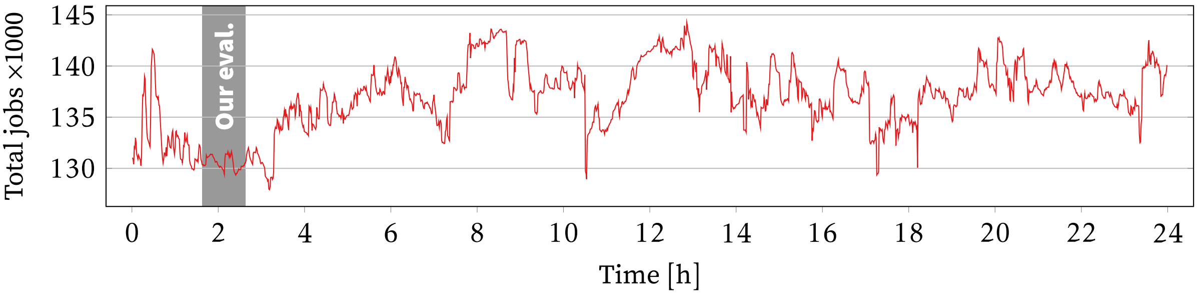

Figure 5.4.: Borg trace: concurrently running jobs during the first 24 h.

Figure 5.4 shows the number of jobs that were active in Google's cluster at any point in time during the first 24 h of the trace. Given the size of Google's cluster, we have to scale down the trace before being able to replay it on our own cluster setup. We scale the trace down along two dimensions: duration and number of jobs.

Figure 5.3 shows the cumulative density function (CDF) of the duration of jobs found in the trace. All jobs last at most 300 s. Hence, 1 h is sufficiently long to properly stabilize the system. Instead of considering the full 29 days recorded in the trace, we use a 1-hour subset ranging from 6480 s to 10 080 s (highlighted in gray in figure 5.4). This slice of trace, while being the less job-intensive in terms of concurrent jobs for the considered time interval, still injects an intensive load on the cluster. To fit our small cluster, we sample every 1200th job from the trace, to end up with a number of jobs big enough to cause contention in the system, but that does not clutter it with an incommensurable amount of jobs.

5.5.3. Matching trace jobs to deployable jobs

After processing the trace file, we get a timed sequence of jobs with their advertised and effective memory usage. In order to materialize this information into actual RAM or EPC usage, our jobs are built around containers that run STRESS-SGX (*cf.*, section 4.2). Normal jobs use the original virtual memory stressor brought from STRESS-NG, while SGX-enabled jobs use

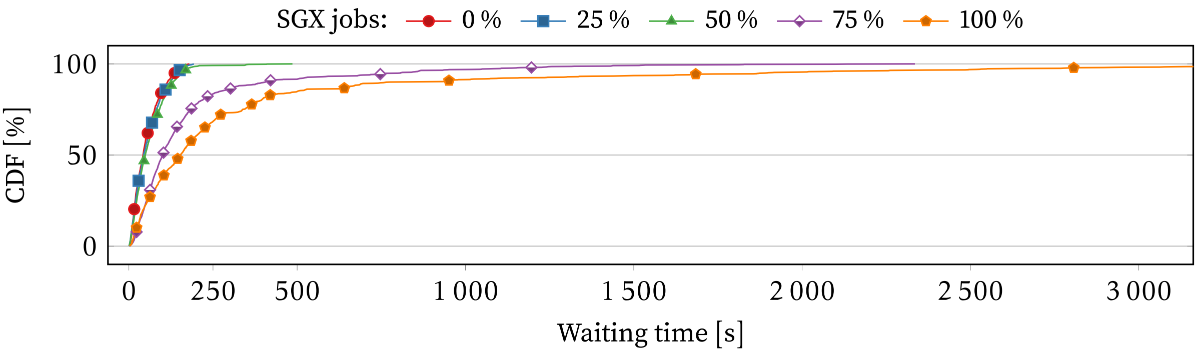

Figure 5.5.: CDF of waiting times, using varying amounts of SGX-enabled jobs.

the topical EPC stressor. We specify parameters to allocate the right amount of memory for every job, in accordance with the values reported by the trace.

5.5.4. Scheduler evaluation

We evaluate our SGX-aware scheduler by replaying the trace described hereinabove. All performance metrics are directly fetched from Kubernetes.

Figure 5.5 shows the CDF of waiting times observed by jobs before their execution. The waiting time refers to the period between the submission of the job to the orchestrator, and the instant when the job actually starts on a given node of the cluster. In this experiment, we use the *binpack* scheduling strategy.

As expected, the run that only uses standard memory (no SGX jobs) experiences relatively low waiting times. On the other side, the pure SGX run waiting times go off the chart, due to much higher contention conditions. The longest wait observed by a job is 4696 s, more than the total task duration given in the trace. When 25 % to 50 % of the jobs are SGX-enabled, waiting times are really close to the ones observed with a fully-native job distribution. This shows that incorporating a reasonable number of SGX jobs has a negligible impact on the scheduling. Notably, we expect real-world deployments to include small percentages of jobs requiring SGX instructions.

Figure 5.6 depicts the waiting times observed in relation to the amount of memory requested by pods, using the *spread* and *binpack* strategies. There are two rows of labels in the x-axis. The top row with smaller values is applicable to SGX jobs while the bottom row is applicable to standard jobs. All values are extracted from the same run with a 50 % split between standard and SGX jobs. The error bars are computed using the 95 % confidence interval. We can observe that the *spread* strategy is consistently worse than *binpack*. Binpack also seems to handle bigger memory requests better. SGX jobs show similar waiting times compared to standard jobs, save for one outlier in the *binpack* plot. This shows that our scheduler works well with both types of jobs.

Finally, figure 5.7 shows the aggregated turnaround time for all jobs. This metric refers to

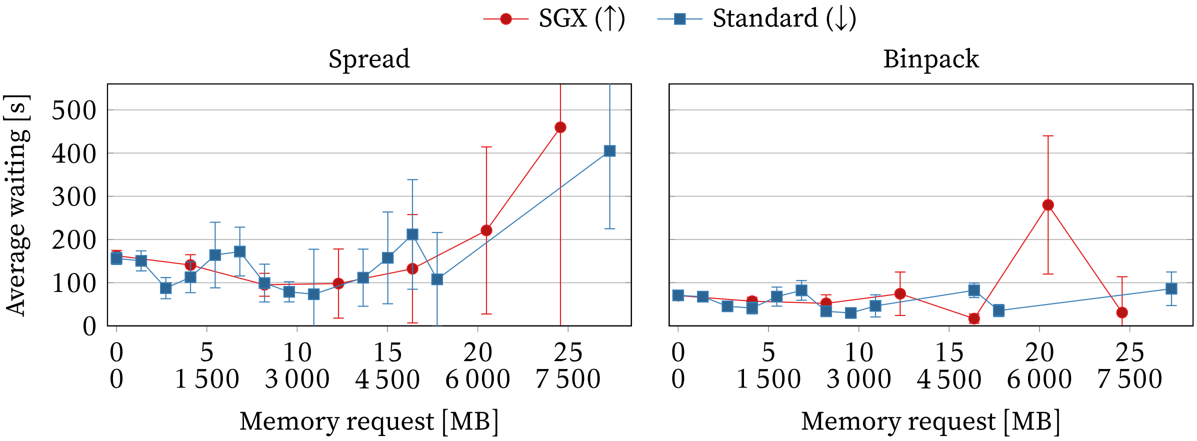

Figure 5.6.: Waiting times for SGX and non-SGX jobs, using *binpack* and *spread* scheduling strategies, depending on the memory requested by pods.

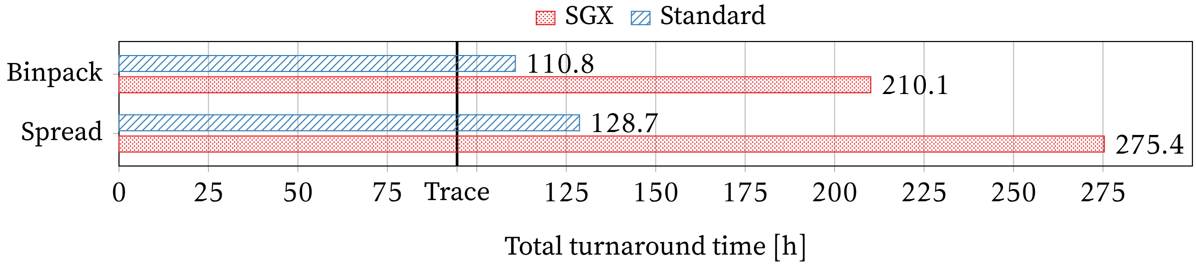

Figure 5.7.: Sum of turnaround times for all jobs sent to the cluster, compared with the time reported by the trace.

the duration elapsed between the instant a job was submitted to the moment when the job finishes and dies, summed for all jobs. The black bar labeled “Trace” represents the total turnaround time recorded in the trace. The difference with the other results highlights the total waiting time for each of the different settings. We use runs that only contain one type of job (either all SGX or regular jobs). As we noted in figure 5.6, the *binpack* strategy, in this specific setting and portion of the trace, achieves the best result (shorter turnaround time). When using the *binpack* strategy, SGX jobs need slightly less than twice the time of their non-SGX counterparts. The total waiting time difference between the two kind of jobs is above this ratio, but the impact on the total turnaround time is limited to some extent. Although a more in-depth evaluation of the trade-offs between the *binpack* and *spread* scheduling strategies would permit a more comprehensive understanding of our setting, we believe that individual workload characteristics are the key factors when selecting a placement strategy.

Our decision to choose a multiplier of 32 GiB for standard jobs and 93.5 MiB for SGX-enabled jobs considerably affects the performance difference between standard and SGX jobs. Indeed, our whole cluster has 2×93.5 MiB $=$ 187 MiB of EPC memory compared to a total amount of 2×64 GiB $+ 2 \times 8$ GiB $=$ 144 GiB of regular system memory. This represents a difference of almost 3 orders of magnitude (788×) between the two kinds of memory, whereas the difference

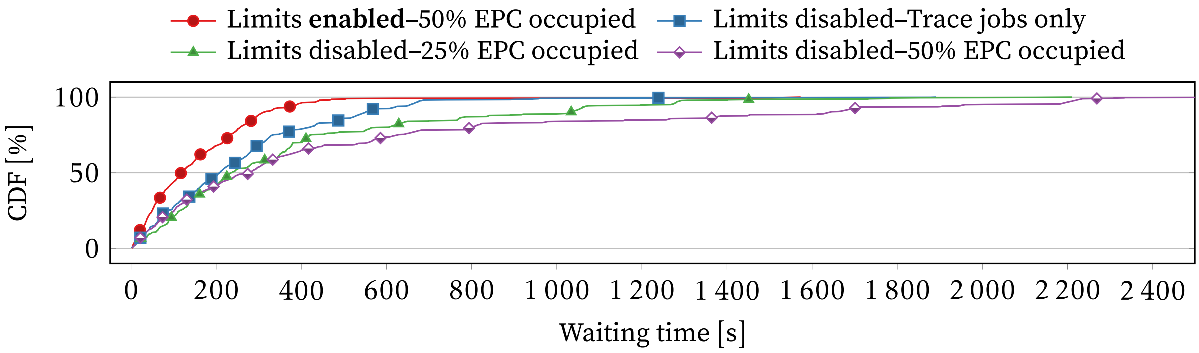

Figure 5.8.: Observed waiting times when malicious containers are deployed in the system, with and without usage limits being enforced.

between the scaling multipliers is only half of that (350×). Therefore, in relative terms, SGX jobs have close to 2× less memory at their disposal compared to standard jobs. As later highlighted in section 5.5.7, doubling the amount of memory drastically improves performance, which explains the performance gap. We observe nonetheless that using a 50 % split between standard and SGX jobs yields acceptable performance for both kinds of jobs.

5.5.5. Impact of resource usage limits

One of the feature of our system is the strict enforcement of limits regarding per-container EPC memory consumption. Resource usage limits are declared by the users themselves, and therefore could be inaccurate with regard to the real usage made by their containers. We identify incentives to lead users into truthfully declaring resource usages: if the user declares too high a limit for their container, then the infrastructure provider will charge them for the additional resources. On the other hand, declaring too low resource usages will lead to the container being denied service due to the enforcement of limits. To show the potential damages that users could cause without this mechanism, we ran the same experiment as before, but we add so-called malicious containers to the system. We deploy as many of them as there are SGX-enabled nodes in the cluster. The *modus operandi* of these containers is to declare 1 page of EPC as limit and request in their pod specification, but actually use way more: up to 50 % of the total EPC available on the machine they execute on.

The results, as presented in figure 5.8, show that, without strict limits being enforced, most honest containers in the system suffer from longer waiting times. Obviously, as the allocation size made by malicious containers increases, the effects suffered by honest containers grow as well. Fortunately, we clearly see that enforcing limits on memory allocations annihilates the efforts of the malicious containers. The reason why the run with malicious containers and limits enabled is better than the one that is just following the trace is because some jobs in the Google Borg trace actually try to allocate more memory than they advertise (44 jobs out of 663 show this behavior). When we strictly enforce memory limits, these jobs are immediately killed after launch by our modified SGX kernel module.

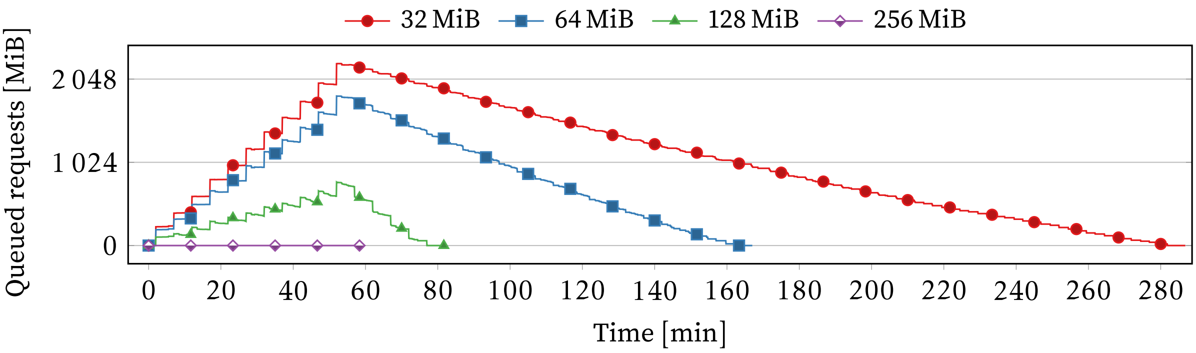

Figure 5.9.: Time series of the total memory amount requested by pods in pending state for different simulated EPC sizes.

5.5.6. Compatibility with SGX 2

In 2021, Intel released processors based on the Ice Lake microarchitecture [87]. These processors support an extended instruction set, often referred to as SGX 2. The most important feature that these new instructions introduce is dynamic EPC memory allocation [126]. Enclaves can ask the operating system for the allocation of new memory pages, and may also release pages they own. Contrary to SGX 1, these operations can also be done during their execution. Another considerable improvement is the much larger EPC region, growing the old maximum of 128 MB in SGX 1 to 512 GB per socket in SGX 2.

Unfortunately, this new hardware was not available at the time of our experiments. Nevertheless, as far as our scheduler is concerned, we believe that only minor changes need to be performed to fully take these new possibilities into account. Provided that Kubernetes nodes are deployed on SGX 2-compatible hardware, we think that our solution will work out-of-the-box. The scheduler already uses up-to-date measurements regarding EPC usage to come up with scheduling decisions. Even when using SGX 1, variations of EPC usage can already happen if a container launches multiple enclaves whose life-cycles are not harmonized. The only part of our system that we have identified as not yet SGX 2-ready is our implementation of resource usage limits in the SGX kernel module. We believe that the effort required to port it to the new revision of SGX is modest.

5.5.7. Simulated evaluation

Through simulation, we evaluate how SGX 2 would impact the performances of our scheduler. We use the exact same algorithms and behavior as our concrete scheduler. That allows us to operate with various EPC sizes, including those that are not supported by the SGX hardware that we have access to.

We use the same trace as for our concrete evaluation (*cf.*, section 5.5.1). Figure 5.9 shows the sum of requested memory by pods in pending state during the replay of the trace. Looking at the extremes, we can notice the total absence of contention when the EPC accounts for 256 MiB, finishing the batch execution in 1 h, exactly as recorded in the trace. Conversely, the trace takes 4 h 47 min to complete when the maximum EPC memory usage is 32 MiB. In between, a 64 MiB EPC would allow the trace execution to finish after 2 h 47 min. For 128 MiB, the maximum EPC limit of SGX 1 processors, the batch would conclude after 1 h 22 min.

This experiment, despite being dependent on this particular job trace, puts in evidence the benefits of having bigger protected memory sizes. Even having 256 MiB would dramatically improve run times. As we now know, SGX 2 supports much larger EPC sizes, and consequently makes SGX more appealing to cloud providers.

5.6. Summary

This chapter presented SK8S, a novel orchestrator for containers running on heterogeneous clusters of servers—a subset of them supporting Intel SGX. The technology allows users to deploy software in the cloud without having to trust their provider, allowing them to trustfully process sensitive data. In previously-existing cloud deployments, customers had to blindly trust their provider not to pry into their jobs nor tamper with them. Conversely, SGX-aware providers had to trust that their customers do not overstep their share of EPC, as they had no means to limit it. Thanks to our novel work, providers can now effectively supervise SGX-related resources by enforcing EPC allocation limits, thence rendering SGX cloud deployments practical.

The challenge in such deployments is to schedule containers with security requirements to SGX machines in priority, which are scarce, while at the same time carefully monitoring their usage of SGX resources. In particular, when exceeding the limited memory capacity of SGX 1 enclaves, performance starts degrading significantly. It is therefore important not to overload SGX machines with too many resource-demanding containers.

To ensure proper monitoring of low-level SGX metrics, we extended the SGX Linux module to gather statistics about the SGX runtime and feed them into the orchestrator, based on Kubernetes. We developed a complete prototype that we openly release [172], deployed it in a private cluster, and conducted a detailed evaluation using Google Borg traces. Our findings reveal that the scheduler must carefully take the EPC size into account to reduce the overall turnaround time. Furthermore, we observed that when half of the jobs in the workload are SGX-enabled, there is virtually no impact on general performance.

Chapter 6.

Scalable privacy-preserving group file sharing

6.1. Introduction

Delegating data storage to cloud services is an efficient method for organizations to cut and adapt costs. However, cloud service providers cannot be fully trusted [22]. A way to overcome the issue is to cryptographically encrypt the data before sending it to storage providers. Data owners may also want to only grant access to well-defined groups of users to create and consume that data. Likewise, cryptographic access control mechanisms can be used to enforce that only valid users can access the encryption keys and, consequently, the data.

Sometimes, the identity of users is also sensitive and needs to be protected. Consider for example military organizations that define access groups based on security clearances. Besides protecting the information that is specific to a clearance level (*e.g.*, confidential, secret and top secret), users sharing the same clearance level do not know each other. Another use-case are virtual data rooms (VDRs) [127] used for exchanging confidential documents during business acquisitions. They not only need to enforce a high-level of access control, but also to protect stakeholders' identities.

Existing research covers cryptographic access control mechanisms that ensure data *confidentiality* and *authenticity* [55, 145, 36], but not *anonymity*. These systems rely on public key cryptography to map user identities and symmetric cryptography to protect the actual shared content. On the contrary, confidential systems focusing on group communication offer anonymity guarantees by group key exchange methods [9], requiring all active group members to be present and participate in a multi-phase protocol (*e.g.*, Diffie-Hellman (DH)) each time a key is derived. Such an approach is indeed suitable for instant group communication, but impractical for asynchronous file sharing.

Theoretical anonymous file sharing extensions have been hypothesized [19, 122] without ever turning into functional systems.

The need for anonymous sharing of confidential content was straightforwardly addressed by GNU Privacy Guard (GnuPG) [165]. Their approach is to drop any public key mapping from the resulting ciphertext, eliminating any reference to the recipients' identities. The main drawback of this solution happens at decryption time, as recipients need to perform numerous asymmetric decryption trials until the portion of the ciphertext that matches their private key is found. As pointed by our preliminary benchmark (*cf.*, table 6.1 on page 47), GnuPG works well for groups of few users but quickly becomes impractical for larger ones.

Taking all the above into account, using TEEs such as Intel SGX as a building block for anonymous sharing systems seems like a compelling idea. However, SGX 1 comes with side costs, notably the overhead associated with the transition between trusted and untrusted modes (*cf.*, section 4.3.4), as well as page swapping when exceeding the limited memory size of the EPC (*cf.*, section 4.3.3). Moreover, we need to consider that some participants in an anonymous sharing scheme may not have access to a TEE, *e.g.*, mobile users or Internet of things (IoT) devices in the case of automated systems.

We propose an anonymous access control scheme that only leverages Intel SGX for a narrow scope of the deployment: enforcing anonymity during the publishing operation, *i.e.*, *writing*. Our scheme does not require a TEE when performing the *read* operation. By leveraging TEEs, we can circumvent assumptions of state-of-the-art theoretical anonymous sharing schemes [19] and considerably improve the performance of cryptographic operations. To demonstrate the feasibility of our solution, we propose a scalable system design leveraging micro-services that can elastically scale depending on the access control and data content workloads.

6.2. Motivation

We provide an overview of the assumptions and security objectives of file sharing systems that guarantee data confidentiality and user anonymity.

6.2.1. Model

We target file sharing between *users* which can be humans or software agents. We consider that users are uniquely identified within the premises of an organization. Users are organized into uniquely-identifiable *groups* by organization-specific considerations and policies. We consider a separation between the group access control and group content management by both functional and threat factors. Group access control represents group memberships operations and is performed by *administrators*. Administrative operations consist in adding and removing users from groups. Group content management represents creating and consuming files by group members. A user can hold one or both roles of *writer* and *reader* within one or multiple groups. The remote storage is a typical cloud object storage that can store uniquely-identified large binary objects (*e.g.*, Amazon Simple Storage Service (S3) [7]).

Exemplifying use-case

Virtual data rooms (VDRs) [127] enable a tightly controlled exchange repository of electronic documents for company mergers and acquisitions (M&As). Thanks to VDRs, the seller, supporting parties assisting the seller, and acquisition bidders can confidentially exchange documents (*e.g.*, terms, valuation) through an untrusted remote storage medium. The seller acts as *administrator* and enforces access control. Active user roles are constituted by *writers* (the seller and supporting parties) and *readers* (the bidders). As enforced by confidentiality agreements, supporting parties operate under the umbrella of the seller, and remain unidentifiable from each other. Similarly, the seller can conceal the identity of bidders among themselves.

As such, *inner* anonymity guarantees need to be enforced within the *writers* (supporting parties) and *readers* (bidders) groups, while *outer* anonymity needs to withstand against any actor who is not involved in the M&A process. Additionally, any withdrawing bidder or misbehaving supporting party can be *revoked* by the seller, therefore becoming unable to access the document corpus.

6.2.2. Security objectives

We specify four high-level security properties for confidential and anonymous file sharing systems.

O1 *Confidentiality and authenticity.* The secrecy of the content of shared files is exclusive to the group members. Recipients should be able to check the integrity and origin of shared content.

O2 *Forward secrecy.* The compromise of a group secret should not compromise past sharing sessions within the same group.

O3 *Recipients privacy.* No recipient except the group administrator should be able to infer the identities of other recipients (*i.e., readers*).

O4 *Sender privacy.* No recipient except the group administrator should be able to infer the sender's (*i.e., writer*) identity.

6.2.3. Threat model

Revoked users and users external to the system behave arbitrarily. They try to discover shared content and group members' identities. To do so, they can intercept, decipher and alter exchanged messages (*i.e.*, Dolev-Yao [46] adversarial model).

User anonymity is not only endangered by external adversaries, but also internally by peer group members. As such, we consider that active users that can rightfully decrypt group content are able to launch attacks with the goal of inferring peer members identities. To do so, they can make use of unlimited attack trials accordingly to their adversarial strategy. We therefore consider that the proposed solution should satisfy the strong security notion of indistinguishability under adaptive chosen ciphertext attack (IND-CCA2). A solution that fulfills such guarantees also satisfies weaker security notions of non-adaptive chosen ciphertext (CCA1) or chosen plaintext attacks (CPA) [20].

The storage provider behaves in an *honest-but-curious* manner. As such, it can try to observe the incoming and outgoing data flows with the goal of discovering the actual content and the identity of the users accessing the data, all while providing service. In order to break the confidentiality guarantee, revoked users can collude with the cloud storage to discover content created after their revocation.

Finally, our privacy model enforces the anonymity guarantee only with respect to user identities. We consider hiding the size of groups, how often members communicate and the size of the content that they exchange as out-of-scope.

6.3. Related work

This section discusses related work and open challenges in the domain of cryptographic cloud storage and access control.

6.3.1. Cryptographic cloud storage and access control

In recent years, a number of storage and sharing system designs have been proposed to mitigate the lack of trust in cloud providers. DepSKY [21] proposes an object store interface that can be used on the client side to encrypt and redundantly store ciphertext on multiple untrusted storage systems. The encryption keys are split by using a secret sharing scheme [155] and dispersed over multiple storage systems that do not collude with each other. SCFS [22] extends the client-side encryption and cloud redundancy of DepSKY by using a trusted metadata coordination service that also encapsulates access control.

Some systems follow a different avenue by cryptographically enforcing access control using key enveloping. Also referred to as *hybrid encryption* [55], the technique consists in encrypting data with a symmetric key that is then itself encrypted with public key encryption. For example, CloudProof [145] proposes a client-side encrypted cloud storage that solves access control by using *broadcast encryption* [27] to envelope two keys: the first is used for decryption (*i.e.*, *reads*) and the second one for signing (*i.e.*, *writes*). In contrast, REED [121] uses *attribute-based encryption* [23] to envelope the symmetric keys that are protecting de-duplicated content. However, the key-enveloping technique was argued by Garrison *et al.* [55] as being impractical when target usage conditions are highly dynamic. A previous contribution of ours, IBBE-SGX [36] demonstrates that the approach can be implemented within dynamic conditions when leveraging TEEs. Yet, none of the above constructions considers an enriched threat model for preserving both confidentiality and anonymity.

6.3.2. Confidential messaging systems

Encrypted messaging systems share a common initial phase with our cloud file sharing model by requiring the construction of a group key that protects group communication. Popular messaging systems (*e.g.*, WhatsApp, Threema, Signal) use a DH group key agreement and derivation [153]. Such protocols require all active participants to contribute to the creation of the group key, albeit without providing anonymity guarantees. Pung [9] uses private information retrieval in conjunction to a group DH key derivation, thus achieving anonymity. Such a mechanism is different from our target model, in which active users do not need to participate in the creation of the group key, no matter the number of groups they belong to.

Table 6.1.: GnuPG operations duration in *hidden recipient* mode.

Group size	Encryption time [s]	Decryption time [s]	Envelope size [kB]
10	0.13	0.6	5.3
100	0.7	5.8	16.5
1000	12	60	129

Pretty Good Privacy

Among practical systems, the popular Pretty Good Privacy (PGP) [189] program—and its free alternative GnuPG—addressed the anonymity criteria with a simple solution. In *hidden recipient* mode [166], after performing the symmetric encryption of the content and public key encryptions of the symmetric key, all the public key mappings are dropped from the resulting ciphertext. As such, an outside adversary cannot directly infer the public keys of the recipients. At decryption time, as the recipients have no pointer to their key-envelope ciphertext fragment, they need to perform several private key decryption trials until they succeed ($\frac{n}{2}$ trials on average, where n is the group size). Table 6.1 presents results of a simple benchmark of GnuPG (version 1.4.2) in *hidden recipient* mode. One can observe that encryption and, more notably, decryption have an impractical cost of 12 and 60 s respectively for groups of 1000 members. Moreover, the implementation of *hidden recipient* is reputed as insecure against chosen ciphertext attacks [19], our targeted threat model.

6.3.3. Anonymous broadcast encryption

The theoretical problem of devising a cryptographic scheme that can guarantee both confidentiality and anonymity is referred to as anonymous broadcast encryption (ANOBE). Theoretical research literature proposes a number of such schemes, however without assessing their practicality within real systems.

The *private broadcast encryption* mechanism proposed by Barth, Boneh and Waters (hereafter BBW) [19] achieves inner and outer anonymity in addition to providing IND-CCA guarantees. Their construction extends the public key enveloping model of PGP by incorporating strongly unforgeable signatures such that an active attacker who is member of the group cannot reuse the envelope to broadcast arbitrary messages to the group. Moreover, to decrease the number of decryption trials, they propose the construction of publicly-known labels, unique for each member of every single encryption operation, by relying on the security assumption of DH. The ciphertext fragments created by the key enveloping process are therefore ordered by their label. During decryption, after reconstructing the label, the user can seek the corresponding ciphertext fragment in logarithmic time before performing a single asymmetric decryption. The scheme was further extended by Libert *et al.* [122] by suggesting the use of *tag-based encryption* [125] to hint users to their ciphertext fragment. To the best of our knowledge, no practical system has integrated tag-based encryption in practice.

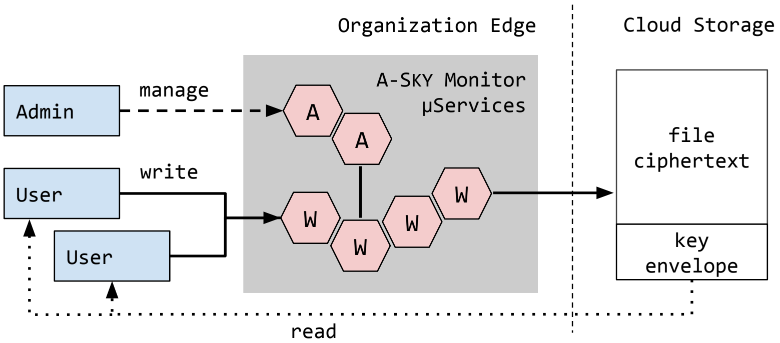

Figure 6.1.: A-Sky solution overview. A-Sky monitor services are AccessControl (A) and WriterShield (W).

As pointed out by our comparison benchmark (*cf.*, section 6.6.1), BBW can handle a key enveloping throughput of only few hundreds of users per second. Such a limitation requires the exploration of alternative constructions that can scale to realistic access control workloads.

6.4. A-Sky

Our solution, **A-Sky**, conceptually relies on two paradigms: a cryptographic key management solution and a data delivery protocol, both designed to target an increased system performance, covering data confidentiality and user anonymity guarantees. We describe our solution by first having an overall look into the proposed architecture. We continue by detailing the design of each system operation. Finally, we briefly discuss the security guarantees of our scheme.

6.4.1. System architecture

A-Sky leverages Intel SGX as a TEE. In order to avoid passing all the operations through a TEE-enabled *monitor*, we propose a design in which only data owners (*i.e.*, *writers*) need to pass through such a proxy. *Readers* anonymously consume confidential content without needing to pass through the TEE-enabled monitor, avoiding potential service time penalties. The benefits of using a monitor exclusively for write operations are numerous. First, the monitor acts as an outbound trusted authority (TA) authenticating all the content passing through. Second, it can mask the identities of data writers. Third, as the monitor executes in a TEE, traditional anonymous key management schemes [19, 122] can be modified to accommodate a new entity of trust for the key enveloping operation, therefore allowing more efficient operations.

Figure 6.1 shows an overview of our solution. The monitor sits in between end-users and the cloud storage and is logically split in two roles. First, the AccessControl service provides a cryptographic mechanism for storing and enforcing access control to the data by performing cryptographic key management. Second, upon successful access verification, the WRITERSHIELD acts as an outgoing proxy for write operations. As system scalability is of paramount importance, the two logical entities can independently adapt to current load.

Key management

The first building block of our design is a cryptographic key management solution. Data owners have to write content through the TEE-enabled monitor such that only authorized readers can decrypt the data, all while having anonymity guarantees. In traditional anonymous key sharing solutions [19], a TA performs two operations: setting up the key management system and extracting user private keys. The operations of key enveloping together with content encryption and decryption are performed by end users. As such, public key cryptographic primitives are employed so that users can cryptographically protect content for other users, whose identities are represented by public keys.

In contrast, our model requires that the TA performs the key enveloping operation in addition to setting up the system and extracting user keys. This change of assumption, which is allowed by using a TEE as an outgoing monitor, allows us to use a much simpler cryptographic construct to achieve the same result as traditional schemes. Concretely, the TA can directly make use of users' secret keys during the key-enveloping operation. As such, this shared secret between the TA and end users opens the way to the use of *symmetric* rather then *public key* cryptography, and therefore benefit from the performance advantages of the former, *e.g.*, hardware acceleration and smaller ciphertexts. BBW requires the construction of a signature key-pair per each key enveloping; with our new assumptions, we can leverage the signature of the TA. Moreover, the shared secret between users and TA allows to construct efficient key de-enveloping methods that increase the performance of the decryption operation performed by end users.

Data delivery protocol

A-SKY allows users to write encrypted shared content through the WRITERSHIELD service, which acts as a proxy. The service will check with the ACCESSCONTROL service whether a user has the permission to write in a given group. Being the case, it authenticates the outgoing content and does the writing itself. We can therefore securely store cloud storage credentials in the WRITERSHIELD service.

TEE trust establishment

Before relying on any service of the A-SKY monitor, it is necessary to validate that the service is running on a trustworthy Intel SGX platform, and that the instances of the ACCESSCONTROL and WRITERSHIELD services are genuine. This validation phase is performed by *administrators*, who are considered trusted in our model. They retrieve the attestation packages and check that the received digests are identical to known ACCESSCONTROL or WRITERSHIELD digests. They then contact Intel Attestation Service (IAS) to validate that the quote signature is indeed genuine. Upon a successful verification, administrators rely on the remote attestation functionality to establish a secure channel using a DH key exchange with both the

AccessControl and WriterShield services [40]. This secure channel is used for subsequent access control operations, such as user creation, addition or removal of members from groups. Besides, administrators are able to securely provision cloud storage credentials to the WriterShield service along with a long term signing key $sign_{TA}$ that is employed on all future content.

6.4.2. Operations design

We formally define the operations of the AccessControl and WriterShield services. We use the following notations:

$E_k(p) \rightarrow c$ denotes the symmetric encryption of plaintext p to ciphertext c using key k.

$D_k(c) \rightarrow p$ is the inverse operation to E_k: symmetric decryption.

$AE_k(p) \rightarrow (c, t)$ represents an authenticated encryption (AE) (*e.g.*, advanced encryption standard (AES) Galois counter mode (GCM)) that, besides the aforementioned symmetric primitives, produce an authentication tag t proving the integrity of the ciphertext c under the key k.

$AD_k(c, t) \rightarrow \{p, \perp\}$ is the inverse operation to AE_k. An output of $\perp$ means that the integrity of c is compromised.

$S_{pri}(p) \rightarrow \sigma$ defines the asymmetric digital signature of p using private key pri.

$V_{pub}(p, \sigma) \rightarrow \{true, \perp\}$ denotes the verification of a digital signature S_{pri} using the public key pub counterpart to pri.

$\mathscr{H}$ denotes a one-way cryptographic hash function, *e.g.*, secure hash algorithm (SHA).

$+\!\!+$ denotes the literal concatenation operation.

AccessControl service

The AccessControl service is responsible for storing credentials and membership information, and to enforce them. Its methods are invoked by administrators through the secure channel established upon successfully performing the trust attestation process.

The AccessControl service generates user secret keys. Given a unique user identifier u_{id}, the service constructs a random secret key for the user, to whom it is sent through a transport layer security (TLS) channel. It further exposes methods for group management. Specifically, administrators can create groups, as well as add or remove users from them. Depending on the granted access capabilities, users can hold the roles of content *reader*, *writer*, or both. The AccessControl service captures such capabilities within persistent dictionaries $group^r$ and $group^w$, which store lists of users belonging to each group identifier. Administrators are the only entities that can modify the keys and values of those two dictionaries.

The operation of enveloping an access key for a group of anonymous members is depicted in algorithm 6.1. Given the identity of the writing user, the group unique identifier, and a

Algorithm 6.1.: Linear key enveloping by ACCESSCONTROL.

```
Input: user identity $u_{id}$, group identifier $g_{id}$, symmetric key $k$.
Output: an $envelope$ ciphertext of the access control key.
1: procedure $\mathscr{A}.KeyEnveloping(u_{id}, g_{id}, k)$
2:     $envelope \leftarrow \emptyset$
3:     if $u_{id} \in group^{w}[g_{id}]$ then
4:         for all users $u \in group^{r}[g_{id}]$ do
5:             $u_{sk} \leftarrow keys[u_{id}]$
6:             $(c_k, t) \leftarrow AE_{u_{sk}}(k)$
7:             $envelope \leftarrow envelope \cup \{(c_k, t)\}$
8:     return $envelope$
```

Algorithm 6.2.: File proxying by WRITERSHIELD.

```
Input: user identity $u_{id}$, group identifier $g_{id}$, file ciphertext $\mathscr{C}$, ACCESSCONTROL instance $\mathscr{A}$.
1: procedure $\mathscr{W}.ProxyFile(u_{id}, g_{id}, \mathscr{C}, \mathscr{A})$
2:     if $u_{id} \in \mathscr{A}.group^{w}[g_{id}]$ then
3:         $\sigma \leftarrow S_{sign_{TA}}(\mathscr{C})$
4:         Upload to cloud: $(\mathscr{C}, \sigma)$
```

symmetric key, the algorithm produces a ciphertext *envelope* that can be anonymously de-enveloped. The operation proceeds by first checking that the user has writing capabilities for the group (line 3). If true, the *envelope* is constructed by including the ciphertext and the authentication tag resulting from the encryption of the symmetric key using the secret key of each group member (lines 4–7).

WRITERSHIELD service

As WRITERSHIELD is the sole service possessing the credentials to write to the cloud storage, it constitutes a necessary intermediary when uploading a file. Its main operation is $ProxyFile$ (*cf.*, algorithm 6.2). It verifies that the invoking user has write capabilities for the desired group (line 2). If positive, the content is authenticated by using the long term TA signature (line 3). Finally, both the file ciphertext and the corresponding signature are uploaded to the cloud (line 4).

Users

Users perform two operations: they can share a file with a group (*i.e.*, writing), and read a shared file. The write operation leverages the TEE-enabled monitor. As shown in algorithm 6.3, the user first randomly creates a symmetric key (line 1). They then ask the ACCESSCONTROL service to perform an enveloping for this key (line 2), so that it could be anonymously de-enveloped by any group member. The file itself then gets encrypted using the randomly-generated symmetric key fk (line 3). Finally, the two resulting ciphertexts—the key envelope and the file ciphertext—are concatenated (line 4) and transmitted to the WRITERSHIELD, which is in charge of uploading them to the cloud storage (line 5).

Algorithm 6.3.: User writing file to group.

```
Input: user identity $u_{id}$, group identifier $g_{id}$, file plaintext $P$, AccessControl and WriterShield
    instances $\mathscr{A}$ and $\mathscr{W}$.
$fk \leftarrow Random$ symmetric key
$envelope \leftarrow \mathscr{A}.KeyEnveloping(u_{id}, g_{id}, fk)$
$cipher \leftarrow E_{fk}(P)$
$\mathscr{C} \leftarrow envelope \mathbin{+\!\!+} cipher$
$\mathscr{W}.ProxyFile(u_{id}, g_{id}, \mathscr{C}, \mathscr{A})$
```

Algorithm 6.4.: User reading file using a linear envelope.

```
Input: user secret key $u_{sk}$.
1: $Download$ from cloud: $(\mathscr{C}, \sigma)$
2: if $V_{pub-sign_{TA}}(\mathscr{C}, \sigma) \neq \perp$ then
3:     $envelope, cipher \leftarrow split(\mathscr{C})$
4:     for all pairs $(k_c, t)$ in $envelope$ do
5:         $fk \leftarrow AD_{u_{sk}}(k_c, t)$
6:         if $fk \neq \perp$ then
7:             $P \leftarrow D_{fk}(cipher)$
8:             return $P$
9: return $\perp$
```

Reading files is done by following algorithm 6.4. As previously stated, reading operations do not involve TEEs. The first step is to download the ciphertext package from the cloud storage (line 1), that can then be validated by checking its signature (line 2) that has been appended by the WriterShield. Provided that the signature is valid, the user then splits the package between the key envelope and the file ciphertext (line 3). They then iterate over all envelope fragments, trying to decrypt each of them by using their secret key u_{sk} (lines 4–5). When the decryption is successful, the obtained plaintext is the file encryption key, that the user can use to symmetrically decrypt the file ciphertext (lines 6–8).

6.4.3. Indexing for efficient decryption

Following the methodology of traditional ANOBE schemes [19, 122], we propose a method that can reduce decryption time by circumventing the need to perform several key decryption trials (*cf.*, line 4 of algorithm 6.4) by trading it off for a slight increase in key enveloping time and envelope size. To this end, labels are constructed for each user in the envelope, such that the label can be recomputed by each recipient. User keys are ordered by labels in the envelope, so that each key can be easily located. A single key decryption operation is thus needed.

Traditionally, building such labels required to bear the cost of modular exponentiation [19], or use the theoretical constructs of tag-based encryption [122]. Instead, we have a TA running in a TEE performing key enveloping. It results that the shared secret between users and the TA can also be used to construct efficient decryption labels. We can therefore propose a much simpler and efficient labeling mechanism by relying on the cryptographic hash of the shared secret, *i.e.*, the user secret key.

The indexed variant of key enveloping introduces the creation of labels as the salted hash of the user secret key. A random *nonce* is generated for each key enveloping call to be used as a salt value, publicly included in the envelope. Each label is computed as $\mathscr{H}(u_{sk} \mathbin{+\!\!+} nonce)$. Envelope fragments can therefore be sorted using label values.

On the receiving side, the reading operation first requires to reconstruct the label (same operation $\mathscr{H}(u_{sk} \mathbin{+\!\!+} nonce)$), followed by a binary search of it among the envelope fragments. When the proper label is located, the file key can be retrieved to finally decrypt the file.

The trade-off is an overhead of $O(n \cdot \log n)$ to sort the labels during the key enveloping operation. The gain is reflected during decryption time, where $O(n)$ trials of symmetric decryption are replaced by a $O(\log n)$ binary search and a single symmetric decryption.

6.4.4. A note on revocation

We argue that A-Sky satisfies the *lazy* revocation model [14], where a revoked user can continue to access data created prior to revocation, but becomes unable to access any data created beyond that. An administrator can revoke a user by removing their identifier u_{id} from the $group^r$ and $group^w$ access lists. Later, when new content is published in that group, the revoked user's key will not be included in the envelope, rendering them unable to access the new group key along with newly published content.

6.5. Implementation

This section describes the implementation details of our system.

6.5.1. AccessControl

The AccessControl service is the only stateful component of A-Sky. It is responsible for generating and storing user keys, and for maintaining group membership information. Since it deals with sensitive information, its core runs within SGX enclaves. All external exchanges are encrypted by using TLS connections that are terminated inside the TEE.

We divide the AccessControl service into two sub-components. The first one constitutes the entry-point for service requests. It is developed in C++, for a total of 3000 lines of code. The other sub-component holds users and groups metadata within a replicated database. For this purpose, we use a triple-replicated cluster of MongoDB [131] servers. MongoDB 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 MongoDB client library [130] to run inside an enclave.

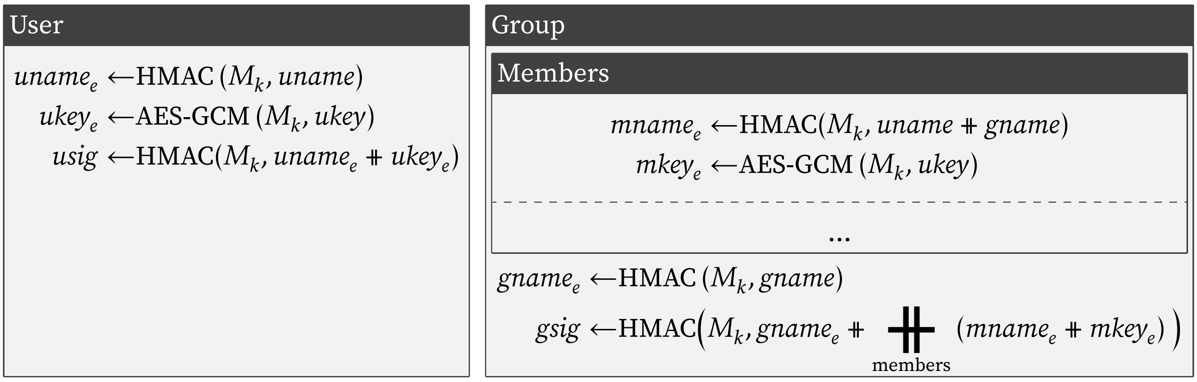
User
$uname_e \leftarrow \text{HMAC}\,(M_k, uname)$
$ukey_e \leftarrow \text{AES-GCM}\,(M_k, ukey)$
$usig \leftarrow \text{HMAC}(M_k, uname_e \mathbin{+\!\!+} ukey_e)$
Group
Members
$mname_e \leftarrow \text{HMAC}(M_k, uname \mathbin{+\!\!+} gname)$
$mkey_e \leftarrow \text{AES-GCM}\,(M_k, ukey)$
...
$gname_e \leftarrow \text{HMAC}\,(M_k, gname)$
$gsig \leftarrow \text{HMAC}\Big(M_k, gname_e \mathbin{+\!\!+} \underset{\text{members}}{\mathbin{+\!\!+}} (mname_e \mathbin{+\!\!+} mkey_e)\Big)$

Figure 6.2.: Data model of user and group documents stored in MongoDB.

As the storage backend runs outside of enclaves, we make sure that every piece of data that we store is either hashed using keyed-hash message authentication codes (HMAC), or encrypted using AES GCM. Accordingly, each replica of the entry-point sub-component is provisioned with a master key M_k at attestation time. We thus guarantee that the entity that hosts the MongoDB cluster cannot infer any information about users or groups (barring the size of each group, which is already leaked in the envelopes).

Figure 6.2 shows how we organize data in MongoDB. We use two collections, one for users and one for groups. Each user is stored once in the users collection and once per group it is a member of. This denormalization prevents the service provider from inferring which groups a user belongs to as the attributes of a given user are hashed or encrypted differently for each representation: we use the name of the group as salt when hashing, and different initialization vectors (IVs) when encrypting. Each document is wholly signed using HMAC signatures to ensure its integrity.

All interactions with AccessControl happen through TLS-encrypted representational state transfer (REST) exchanges formatted in JavaScript object notation (JSON). In order to terminate TLS connections in the enclave, we use a port of OpenSSL for SGX [71], while we slightly modified a C++ library to support JSON [123].

The AccessControl service offers few endpoints. Administrators use it to perform group access control operations. Users, on the other hand, first use AccessControl to retrieve their randomly-generated 256 bit private key. Furthermore, the AccessControl service is in charge of generating key envelopes upon user requests. An envelope contains a file key encrypted several times, once per group member. The file key, as well as the user keys, are 32 B long. We use AES GCM, which generates a tag of 16 B for integrity. Considering the addition of 12 B for the IV, each group member adds 60 B to the envelope.

In order to avoid having to perform $O(n)$ decryption trials, we can index the keys within the envelope (*cf.*, section 6.4.3). First, we generate a nonce for each envelope, that we staple to it. Each user key is then hashed using SHA224, using the nonce as a salt. This adds 28 B to the envelope for each group member. The list of keys is sorted using the hashes as a sorting key.

As a consequence, readers can look for their own key by doing a binary search, decreasing the complexity to $O(\log n)$ comparisons followed by one single decryption.

6.5.2. WriterShield

WriterShield serves two purposes: it protects cloud storage credentials, and hides user identities by proxying their requests to the cloud storage. When forwarding user requests to write files, WriterShield checks with AccessControl that the query comes from a user who has the required permissions to write files.

User requests, including file contents, cross over the enclave boundary. This obviously slows down transmission rates because of content re-encryption and trusted/untrusted edge transitions. Therefore, we have also implemented a different variant where temporary access tokens are given to users, allowing them to upload their content without the aforementioned content needing to enter the TEE. In such case, users are responsible to use appropriate proxies that can conceal the origin of the request. One approach to hide the identities is by using peer-to-peer relay networks backed by enclaves [143]. Also, it is a requirement to only communicate with the cloud storage using encrypted connections. Even though file contents are encrypted, the metadata can leak group information to every entity listening on the network.

6.5.3. Client

As part of our prototype implementation, we develop a full-featured client using the Kotlin language [98]. The client can be set up to operate in all possible configurations of A-Sky: keys in linear or indexed envelopes, writes through WriterShield, or through a standard proxy onto a Minio [129] or Amazon S3 [7] storage backend 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) [158] and Yahoo! Cloud Serving Benchmark (YCSB) [38] frameworks that we use to perform the evaluation of A-Sky.

6.5.4. Deployment

All our components can be independently replicated to provide availability, fault tolerance or cope with the load. Therefore, we package our micro-services as individual containers, which we then deploy using our SGX-aware orchestrator (*cf.*, chapter 5).

6.6. Evaluation

We evaluate the performance and scalability of our solution by way of micro-benchmarks. Then, we use the well-known YCSB [38] 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 GiB of RAM. Additionally, we use 3 Dell PowerEdge R630 dual-socket servers, each equipped with two Intel Xeon E5-2683 v4 CPUs and

Table 6.2.: Comparison of the enveloping and de-enveloping operations of BBW and A-Sky, with linear and indexed envelopes.

	Linear envelope		Indexed envelope	
	Enveloping	De-enveloping	Enveloping	De-enveloping
BBW	$3.3 \times 10^{2}\ \lvert\mathscr{G}\rvert/\mathrm{s}$	$5 \times 10^{3}\ \lvert\mathscr{G}\rvert/\mathrm{s}$	$3 \times 10^{2}\ \lvert\mathscr{G}\rvert/\mathrm{s}$	<4 µs
A-Sky	$1.9 \times 10^{6}\ \lvert\mathscr{G}\rvert/\mathrm{s}$	$2.5 \times 10^{6}\ \lvert\mathscr{G}\rvert/\mathrm{s}$	$1.2 \times 10^{6}\ \lvert\mathscr{G}\rvert/\mathrm{s}$	<4 µs
Difference	3.7 OoM	2.6 OoM	3.6 OoM	0

128 GiB of RAM. One of the latter machines is split in 3 VMs to handle the roles of Kubernetes control plane, Minio server and benchmarking client (when a second client is needed). SGX machines use microcode revision `0x8e` and have the Hyper-threading feature disabled to mitigate the *Foreshadow* security flaw [33]. Communication between machines is handled by a Gigabit Ethernet network. The backend of AccessControl consists in a triple-replicated MongoDB cluster.

When error bars are shown, they represent the 95 % confidence interval.

6.6.1. Cryptographic scheme performance

We start the performance evaluation of A-Sky by isolating and measuring the performance of the underlying cryptographic primitive. We employ the ANOBE scheme defined by Barth, Boneh and Waters (BBW) [19] as a baseline. Our implementation of BBW uses an elliptic curve integrated encryption scheme as per the IND-CCA2 public key cryptosystem used by the original scheme. For both cryptographic schemes, we select key materials (*i.e.*, keys, curve) that meet 256 bit of *security strength* [18]. Moreover, we implement the efficient decryption of BBW as suggested in the paper by relying on the hardness of the DH problem, however in the context of much faster elliptic curve Diffie–Hellman (ECDH). As the content encryption is similarly implemented for the two schemes, we choose to only measure and present the key enveloping and de-enveloping performance. We consider that the user keys are available at the time of the calls.

In table 6.2, we compare the key enveloping and de-enveloping throughput achieved by BBW and A-Sky. We report the number of group members handled per second. We see that, while BBW can only envelope 330 group members per second in a linear envelope, A-Sky raises this figure to more 1.9 million. The considerable speed difference is justified by the performance gap between public key (used by BBW) and symmetric encryption (used by A-Sky) primitives. Likewise, a performance increase of 2.6 orders of magnitude (OoMs) is observed for the de-enveloping operation.

With indexed envelopes, BBW can achieve fast decryption times, but still limited by the enveloping operation. A-Sky, on the other hand, also achieves the same efficient decryption latency, but performs much better with respect to the enveloping operation, with 1.2 million $|\mathscr{G}|$/s, an increase of 3.6 OoMs compared to BBW. Furthermore, A-Sky produces smaller

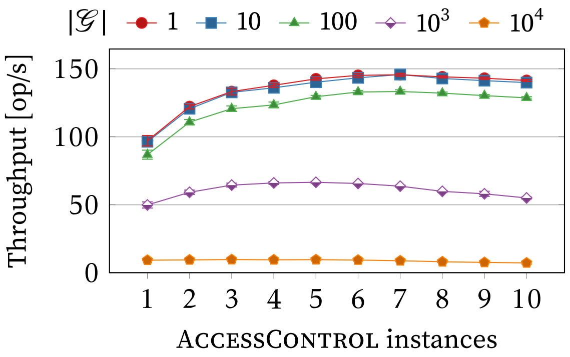

Figure 6.3.: Throughput achieved by AccessControl adding users to groups.

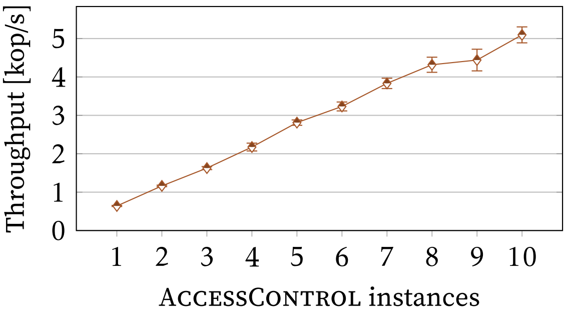

Figure 6.4.: Throughput achieved by AccessControl creating users.

ciphertexts per group member: 60 B for a linear envelope, and 88 B for an indexed one, compared to 126 B and 154 B per member for BBW.

6.6.2. Scalability of AccessControl

We evaluate the throughput of administrative operations when scaling up the number of AccessControl instances. Requests are distributed between instances by exposing a *service* in Kubernetes. The scalability of adding a user to a group or revoking its rights is limited (*cf.*, figure 6.3), as these operations require to perform one read-modify-write (RMW) cycle to check and update the signature of the group document. The larger the group, the more the operation takes time as each signature encompasses every user within the group. This effect could be mitigated by, *e.g.*, batching multiple operations on a given group together. On the other hand, as can be seen in figure 6.4, the operation that creates users scales linearly with the number of AccessControl instances, allowing more than 5000 user creations per second with 10 instances.

Next, we evaluate the scalability of the enveloping operation. In figure 6.5, we observe a close-to-linear trend according to the number of instances, showing that this operation also benefits from horizontal scalability. With groups of 1000 to 10 000 members, the throughput ceases to increase with more than 7 instances as the MongoDB backend becomes a bottleneck. For smaller groups, scalability gains are diminished due to the fixed costs associated with each request (*e.g.*, network connection, REST request, enclave transitions, *etc.*). Nevertheless, increasing the number of AccessControl instances is still beneficial. Additionally, we ran the same experiment with indexed envelopes. For groups of 10 000 users, the throughput is reduced by 6 % to 26 % compared to using linear envelopes. As previously observed, scaling is ineffective for smaller groups where the performance mostly depends on fixed costs.

We also evaluate the latency of the enveloping operation by increasing the throughput until saturation, again using linear and indexed envelopes. Looking at figure 6.6, we notice that for

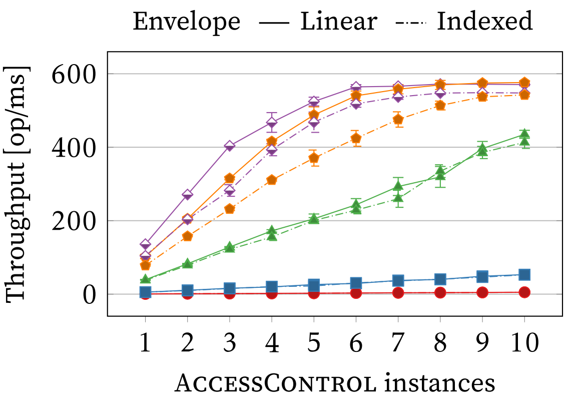

Figure 6.5.: Throughput of enveloping a message.

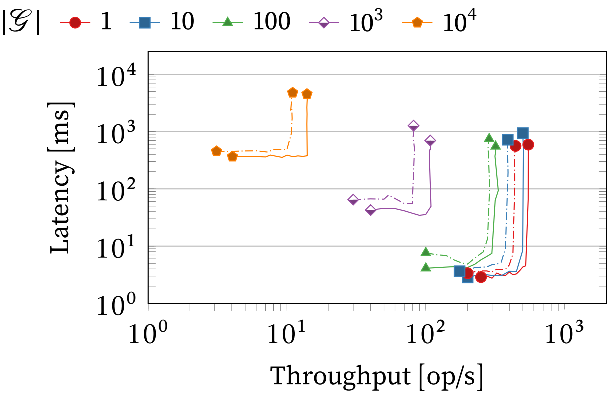

Figure 6.6.: Throughput *vs.* latency plot of enveloping a message.

groups which are larger than 100 users, latency increases linearly according to the group size, while the saturation throughput decreases linearly.

6.6.3. Scalability of WriterShield

To evaluate the performance of WriterShield, we conduct two experiments. In the first one, data written to the cloud is proxied through the TEE. In the second one, WriterShield is only used as a facilitator to obtain temporary access tokens for the cloud storage, with write operations being proxied through an *nginx* server [163] in transmission control protocol (TCP) reverse-proxy mode. In order to establish a baseline, we also write the data directly to the cloud storage service, without any intermediary. Results are shown in figure 6.7. Looking at the bar plot on the left-hand side, we notice that, for files of 1 KiB and 10 KiB, the difference in performance is negligible, whereas bigger files cause more performance degradation when using the token feature. When WriterShield is used to forward data instead (right-hand side), we see that the throughput increases with the number of service instances until it seems to plateau at about the same values as with the tokenized variant. For files of 1 MiB,

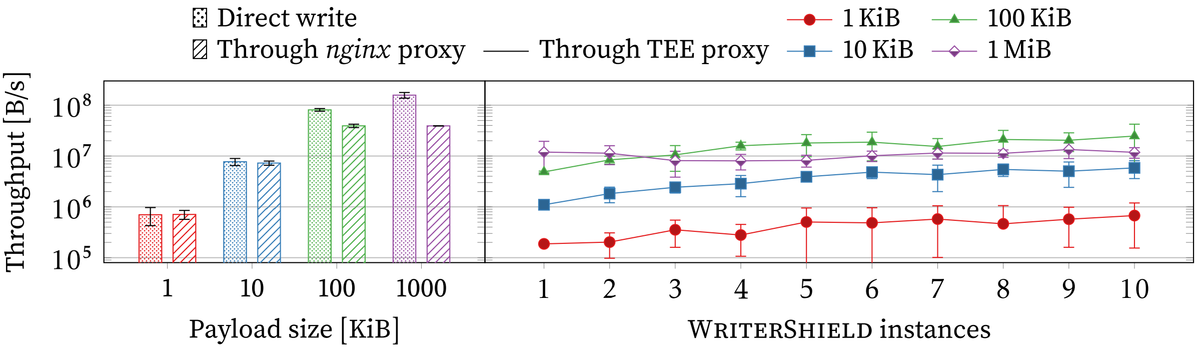

Figure 6.7.: Throughput of writing data to the cloud storage in different ways.

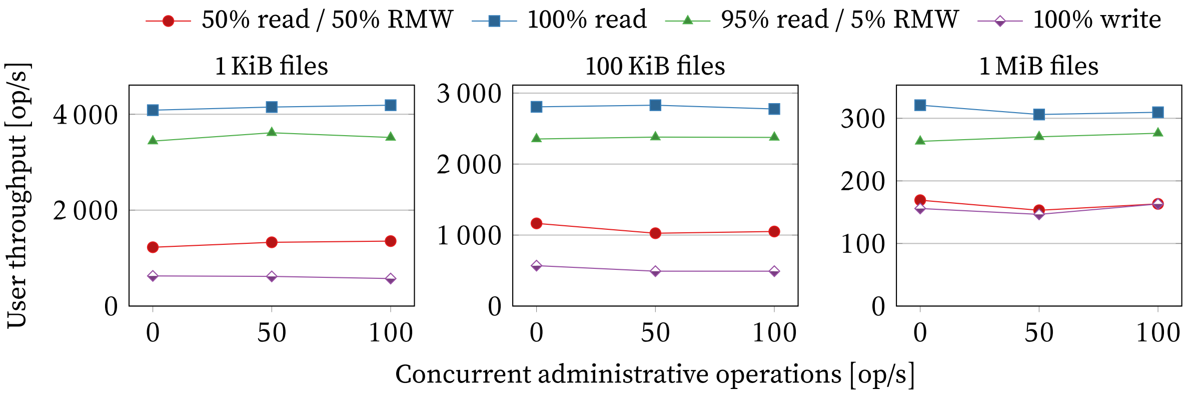

Figure 6.8.: User throughput observed by our YCSB-based macro-benchmark under various conditions.

adding WRITERSHIELD instances shows 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.

6.6.4. Macro-benchmarks

We use YCSB [38] to observe the behavior of A-SKY under different usage conditions that are typical of data serving systems. We implement an interface layer to link the benchmarking tool to the A-SKY client. As our system is not capable of direct-access writes, *update* operations are replaced by RMW operations. We run YCSB workloads *A* (update heavy), *B* (read heavy) and *C* (read only), to which we add an insert-only workload. We consider files of 3 different sizes from 1 KiB to 1 MiB. We simulate 100 000 operations across 64 concurrent users and report upon the user operation throughput. At times, we add a second simultaneously-running instance of YCSB that simulates 8 administrators doing group membership operations. The administrative operations are equally distributed between adding a user to a group and revoking one, so that the size of the user database stays more-or-less constant.

Figure 6.8 shows the results of our experiment. One can notice that the user throughput is not influenced by concurrent administrative operations, as each type of operation involves separate components of our architecture. For small files of 1 KiB, an increasing proportion of writes causes a degradation in performance from 4100 op/s for read-only to 628 op/s for write-only workloads. With larger 1 MiB files, the difference is more nuanced, with a throughput of 320 op/s for the read-only workload compared to 155 op/s for the write-only workload. Therefore, fixed costs are largely dominant when writing small files (*e.g.*, enveloping the file key), but are increasingly amortized for larger file sizes. We can also observe that the throughput in B/s (*i.e.*, multiplying the result in op/s to the file size) is largely superior for larger files, as we have already noticed in figure 6.7. 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.

6.7. Summary

This chapter introduced A-SKY, an end-to-end system that guarantees anonymity and confidentiality of shared content (*e.g.*, files). It leverages TEEs exclusively for the content sharing operation, while TEE capabilities are not required for end-users consuming the shared content. We introduced a novel ANOBE scheme that exploits additional assumptions about the availability of a TEE compared to state-of-the-art schemes, in order to achieve fast and practical performance for its operations. We incorporated the novel cryptographic construction into a scalable system design that leverages micro-services that can elastically scale. Results indicate that our cryptographic scheme is 1000× faster than state-of-the-art ANOBE schemes. An end-to-end system that utilizes our scheme can serve groups of 10 000 users with a throughput of 100 000 key derivations per second per service instance.

Chapter 7.

Offloading network security to untrusted devices

7.1. Introduction

Middleboxes are often deployed within large corporate networks. Their main goal is to filter out various security threats from reaching end-hosts, *e.g.*, through firewalls and intrusion detection and prevention systems (IDPSs). They can also increase performance, *e.g.*, using caching and load balancing. Be that as it may, they need to be efficiently manageable and cost-effective while handling growing amounts of network traffic and ever-increasing network-based attacks.

The current best practice is to deploy middleboxes within networks, despite high infrastructure and management costs. Recent research proposals, instead, investigate the benefits of outsourcing middleboxes to cloud infrastructures [156, 115]. While this reduces maintenance effort and, in turn, cost, deploying critical network functions externally and redirecting sensitive network traffic off-site introduces new potential security risks.

To address those limitations, we propose a new *decentralized* deployment approach in which middlebox functions are executed on client machines at the network edge. Thus, middlebox functions can exploit the potentially idle resources of client machines for processing client traffic. This approach is especially efficient as client traffic constitutes a large fraction of traffic in managed networks [17, 186].

A decentralized deployment model for middleboxes raises two new challenges: (i) clients are entrusted to faithfully execute middlebox functions, and (ii) network administrators need a way to retain control over middlebox functions [17]. While this is uncomplicated to achieve on professionally-managed machines such as servers, today's work practices such as home-office and bring your own device (BYOD) policies where employees retain administrative privileges on their machines mean that the decentralized model is definitely more challenging to envision. Thus far, research proposals have mostly considered host-based deployments of network functions for trusted server machines [156, 17, 115].

We describe **ENDBOX**, a new system for the trusted execution of middlebox functions on client machines. The design of ENDBOX is based on a virtual private network (VPN), which is used to access the managed network from any untrusted one. We enhance the VPN client with support for the execution of trusted middlebox functions through the *Click* software router [108]. ENDBOX intercepts all traffic between the client and the network and ensures that it is processed by middlebox functions executing on the client machine. The functions are guarded by the Intel SGX TEE to protect their integrity.

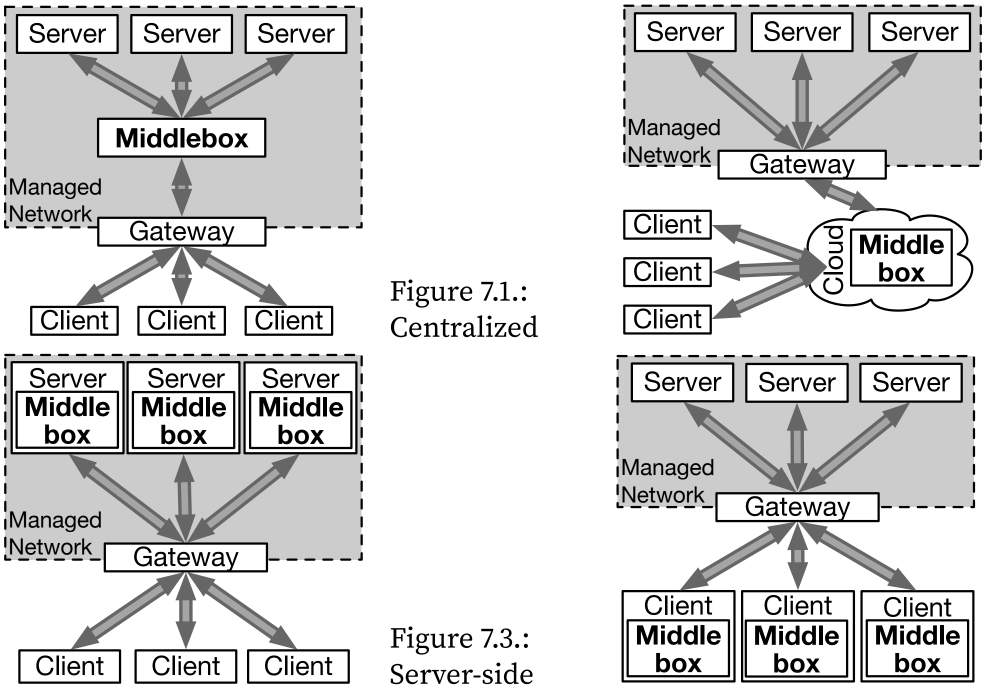

Figure 7.1.: Centralized

Figure 7.2.: Offloaded to the cloud

Figure 7.3.: Server-side

Figure 7.4.: Client-side

Figures 7.1 to 7.4: Different middlebox deployment models.

Another challenge in modern network is the prevalence of encrypted network traffic [73]. We leverage the trusted execution model of EndBox to implement encrypted traffic analysis. In contrast to man-in-the-middle (MitM) proxies, which break end-to-end encrypted sessions, EndBox keeps the encryption keys on the clients, thereby enabling decryption without weakening the overall security.

Despite its decentralized deployment model, EndBox can be reconfigured securely, rapidly, and seamlessly. EndBox forces the clients to always use the latest configuration version.

7.2. Targeted applications

In this section, we clarify the kind of applications that EndBox targets. First, we describe how middleboxes are deployed in today's managed networks, and discuss about the trade-offs of each deployment strategy. Then, we present two scenarios that benefit from the secure client-side middleboxes provided by EndBox. Finally, we discuss our assumed threat model with respect to untrusted clients.

7.2.1. Middlebox deployment strategies

We observe three fundamentally different approaches of deploying middleboxes: (i) centralized deployment as part of the managed network; (ii) cloud-based deployment, and (iii) deployment as part of end-hosts.. In the following sections, we describe them and explain which one we use for EndBox. Later, in section 7.6.2, we measure the latency that a user can expect depending on which middlebox deployment strategy is used.

Centralized middlebox deployments

The most common type of deployment in managed networks it to deploy middleboxes on the link between servers and the network gateway (*cf.* figure 7.1). As middleboxes are diverse and often complex, there is a trend to replace purpose-made hardware appliances with software-based solutions running on top of commodity hardware [109]. With ever-growing network traffic and 100 Gbit/s enterprise links becoming commonplace, this requires scalable software solutions. Since middleboxes are often stateful, *e.g.*, IDPS functions, it is challenging to perform simple horizontal packet-based scaling, as each network flow must be assigned to a single middlebox instance [186]. Centralized middlebox deployments are non-trivial to scale with the number of client machines, resource-intensive and consequently costly [156].

Cloud-based middlebox deployments

In line with the trend of network functions virtualization (NFV) [70], middleboxes can be outsourced to public clouds operated by a third party [156] or private *telco clouds* operated by Internet service providers (ISPs) [161] (*cf.* figure 7.2). Although using public clouds relieves network administrators from having to directly manage the middleboxes, it comes with several downsides: (i) in order to be processed in a cloud infrastructure, traffic must be redirected, thereby incurring additional latency; (ii) public clouds can be considered as untrusted infrastructure, and (iii) traffic redirected to the cloud may be filtered or manipulated on the way.

Offloading middlebox functions to private *telco clouds* may incur less latency, and more trust can be given to the infrastructure. However, it still needs substantial investment by ISPs. Withal, cloud-based middleboxes are convenient to manage, but potentially reduce the reliability of managed networks. Concerns regarding security, latency and legality are often used to discard this deployment strategy.

Middleboxes at end-hosts (clients or servers)

The last deployment strategy is to execute middlebox functions on end-hosts, either on servers in a data center (*cf.* figure 7.3) [17], or on clients inside an enterprise environment (*cf.* figure 7.4; our approach) [44]. These approaches benefit from network traffic being processed directly at its source or destination. Scalability is improved as each end-host handles its own traffic. However, fully-untrusted client hosts have not yet been considered by previous efforts, which is the key challenge that EndBox solves. The closest effort, ETTM [44], does consider untrusted end-hosts, but its approach is limited: (i) it provides lower security guarantees as it cannot withstand physical attacks; (ii) it relies on traffic being correctly forwarded by physical switches, thus extending the TCB of the whole system, and (iii) it builds on an expensive distributed consensus algorithm.

Our goal is to explore a deployment model that targets entirely-untrusted clients and network hardware in order to secure the following benefits: (i) network traffic can be filtered or processed at the source or destination; (ii) processing encrypted traffic does not create vulnerabilities and is practical; (iii) central network devices in a managed network are relieved from having to provide middlebox functions, and (iv) deployments can be made to scale because middlebox functions can use the resources of under-used client machines.

7.2.2. Scenarios

We describe two representative scenarios that benefit from secure client-side middleboxes as provided by EndBox.

Scenario 1: Enterprise network

> A large company seeks to protect their network using middleboxes. Due to the increasing cost of centralized hardware middleboxes, the company decides to offload middlebox functions. It is decided to let client machines execute middlebox functions using EndBox. In line with employees working from remote locations, clients can be connected either to the internal network or join the network remotely using a VPN client.

Scenario 2: ISP network

> An ISP with thousands of customers wants to offer additional protection by performing deep packet inspection (DPI) on network packets. The goal is to protect their customers' machines as well as their own network components from malware, such as ransomware. The product portfolio of the ISP is extended by a data plan that deploys EndBox for network traffic analysis on the client machines of customers. The plan includes a discount to compensate for the allocation of client-side resources.

7.2.3. Threat model

Client machines are typically untrusted, as they elude from the control of network owners. In most companies, a few managed machines can escape from the policies set by the company's information technology (IT) department. Typically, network administrators and developers may possess administrative rights on their machines. In companies that permit their employees to bring their own device (BYOD), we can even assume that most client machines on the managed network will be out of control of the IT department. In the case of the ISP scenario, customers' machines are obviously totally out of control of the network administrator. At any rate, client machines may be vulnerable due to lacking essential security patches or through misconfiguration. These vulnerabilities could be used to circumvent security critical middlebox functions.

We therefore assume that client machines are untrustworthy. We consider that an adversary has unrestricted physical access to the targeted client machine. That includes the operating system (OS), any hypervisor, as well as all its hardware. They have full access over the network card, so can make it send any packet, and can read all incoming packets. Having physical access to the targeted machine, the adversary can read from or write to any memory address.

The adversary can perform denial-of-service (DoS) attacks against the SGX enclave where EndBox runs, *i.e.*, refusing to start or enter it. However, we ignore distributed denial-of-service (DDoS) attacks on the server infrastructure: while malicious clients can collude and send spurious traffic to servers, existing mitigation approaches can be applied [54].

In line with typical assumptions about managed networks, we consider all servers to be under tight central administrative control and thus trustworthy. Client machines are never allowed to directly connect to the managed network because they can be subject to the aforementioned attacks and act maliciously.

In contrast, we assume that users place confidence in who provides the middlebox functions (*i.e.*, their employer or their ISP depending on the scenario). Note that this assumption also holds true in traditional middlebox deployments. We could optionally weaken or remove this assumption of trust towards the provider by enabling users to enforce policies during runtime on SGX enclaves [137].

7.3. Related work

Moving middlebox functions to end hosts has been discussed in previous research efforts, *e.g.*, ETTM [44], Eden [17], SDNFV [186], and by Karagiannis *et al.* [104]. However, most of these solutions assume that end-hosts are trusted, an assumption that cannot be made regarding a client-side deployment like we envision.

One notable exception is ETTM [44], which relies on a TPM. This approach is inflexible because it only supports attestation at bootstrap time and lacks integrity checks during execution. Most importantly, it does not protect against malicious users with physical access to the machine as EndBox does through the use of Intel SGX. Further, ETTM is impractical because the entire hypervisor and physical network hardware need to be part of the TCB. While assuming that the network hardware can be trusted is credible in an enterprise setting, it is infeasible in our ISP scenario. Finally, the design of ETTM follows a distributed approach that leverages Paxos [114] to reach consensus, but Paxos does not scale well [178], induces high latency, and is not applicable to mobile nodes with an unstable connection.

Other proposals such as Eden [17] rely on specialized hardware on end-hosts to implement middlebox functions. While these solutions can achieve higher performance than EndBox, their hardware requirements exceed those of regular laptops and desktops.

Middlebox functionality can be entirely moved to the cloud [156, 115, 185]. This avoids the risk of users mounting physical attacks and can provide great scalability. These benefits, however, come at the cost of higher latency due to traffic redirection. Further, outsourcing traffic processing brings new security risks as well as privacy and legal issues.

Executing middlebox functions inside SGX enclaves has been proposed before [48, 41, 113, 168]. Contrary to ENDBOX, these systems are not designed to be deployed on clients. Instead, they execute entire middleboxes or specific functions in the cloud to guarantee the integrity and confidentiality of network traffic.

The following four proposals also target the problem of executing middlebox functions on encrypted traffic. BlindBox [157] presents an encryption scheme to perform a limited set of computations on encrypted traffic, but at a much lower cost than traditional homomorphic encryption. Both mcTLS [135] and mbTLS [134] propose a way to encrypt packets in a way that allows middleboxes to decrypt them. This is done by generating different cryptographic keys in the case of mcTLS, and by securely forwarding session keys in the case of mbTLS. SGX-Box [72] uses SGX on centralized middleboxes to enable DPI on encrypted network traffic. Similarly to ENDBOX, TLS session keys are securely shared with the enclave.

7.4. Design of ENDBOX

We describe ENDBOX, our system that securely executes middlebox functions on client machines.

7.4.1. Requirements

In accordance with the scenarios and the threat model that we consider (*cf.*, sections 7.2.2 and 7.2.3), we determine that ENDBOX must satisfy the following requirements.

R1 *Enforcement.* ENDBOX will ensure that all traffic between the client and the managed network is always processed by middlebox functions.

R2 *Integrity and privacy.* ENDBOX has to guarantee the integrity of middlebox functions and the privacy of client traffic.

R3 *Flexibility.* ENDBOX needs to enable effortless development of tailored middlebox functions that can target a wide range of use cases.

R4 *Manageability.* ENDBOX will provide seamless management of middlebox functions despite its distributed model.

R5 *Performance.* The performance overhead introduced by ENDBOX should be low and comparable to existing solutions. Moreover, ENDBOX needs to scale linearly with the number of clients.

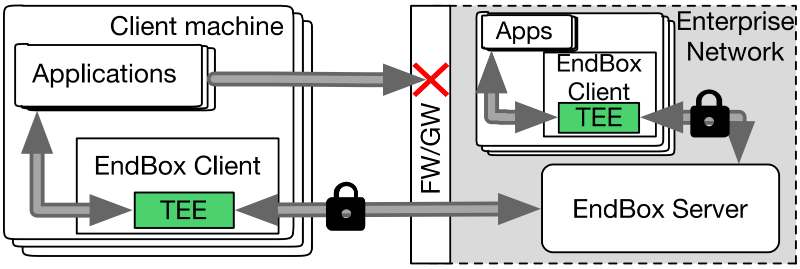

Figure 7.5.: EndBox deployment in an enterprise network.

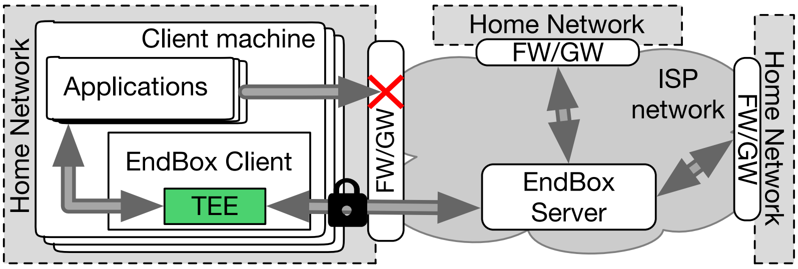

Figure 7.6.: EndBox deployment in a residential ISP network.

7.4.2. Architecture

Figures 7.5 and 7.6 depict two deployments of EndBox, one for each of our two representative scenarios (*cf.*, section 7.2.2). Throughout the figures of this chapter, we use the color green to highlight where code runs in an SGX enclave.

In both deployments, a number of EndBox *clients* connect to an EndBox *server*. In the case of the enterprise network scenario (*cf.*, figure 7.5), clients may be connected to the company network or can connect remotely (*e.g.*, employees working from home). In contrast, in the ISP network scenario (*cf.*, figure 7.6), clients are connected to a home network, which is in turn connected to the ISP network through a filtering gateway.

The use of EndBox is enforced when accessing a managed network because the EndBox server is the only entry point: it only accepts traffic encrypted with the key owned by a correct EndBox client. This ensures that all traffic is processed by EndBox and prevents users from bypassing the middlebox functionality (*R1*).

Middlebox functions are executed in a TEE—our prototype uses SGX enclaves. It guards the endpoint of the VPN communication channel in order to only allow packets that have been screened by middlebox functions through. Encryption keys are injected inside the TEE as part of a secure bootstrapping process so that neither the user of the machine nor software outside the TEE can see them. Packet en- and decryption, as well as any kind of packet processing, happen within the TEE. SGX attestation guarantees that (i) the enclave is initialized with the correct code and data, and (ii) the encryption and decryption of network packets can only occur within the enclave (*R2*).

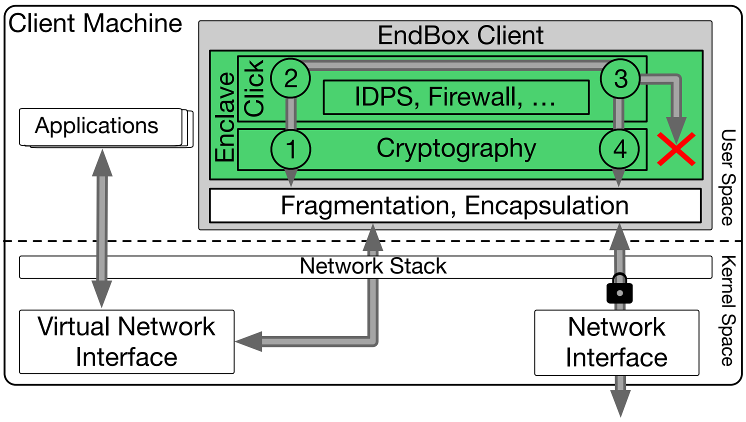

Figure 7.7.: Architecture of the ENDBOX client.

It is possible to implement a wide range of middlebox functions (*R3*), *e.g.*, caching, malware detection, licensing control, and compression. Moreover, operating on encrypted packets becomes practical thanks to the TEE-secured environment.

The ENDBOX server provides a management interface that enables administrators to straightforwardly deploy middlebox configuration changes (*R4*). Configuration updates are disseminated to all connected and (re-)connecting clients, which then asynchronously fetch and apply them. After a configurable grace period, the update is strictly enforced by only letting traffic coming from clients with the latest configuration through.

ENDBOX is designed to induce a low performance overhead, and scale with the number of connected clients (*R5*). We reduce the overhead by limiting the number of enclave mode transitions as much as possible. Scaling is achieved by moving middlebox functions to clients, thus removing load from centralized middleboxes as part of the managed network.

7.4.3. The ENDBOX client

The ENDBOX client architecture shown in figure 7.7 consists of two components: a VPN client and a set of middlebox functions. The VPN client is based on OpenVPN [52] and is partitioned as such: (i) security-sensitive parts (*e.g.*, cryptographic functions and encryption keys) are executed in the SGX enclave to prevent an attacker from gaining knowledge about any secret, and (ii) security-irrelevant parts (*e.g.*, packet encapsulation and fragmentation) are executed outside of the enclave. This partitioning best takes advantage of SGX, taking the limited resources available to SGX 1 enclaves as well as maximizing the performance of ENDBOX (R5).

ENDBOX implements middlebox functions using the Click modular router [108]. It allows us to implement a diverse set of middlebox functions (*R3*). ENDBOX routes all traffic through middlebox functions: OpenVPN hands over all packets to Click for processing right at the VPN boundary. To ensure that all network traffic is intercepted by ENDBOX (*R1*), a client can only connect to the managed network through the VPN.

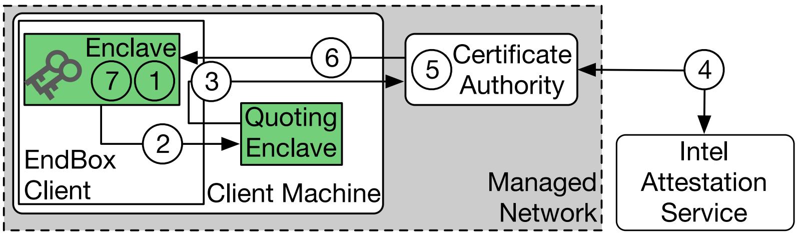

Figure 7.8.: EndBox remote attestation and key management.

When an application wants to send a packet to the managed network, it communicates with the virtual network interface exposed by OpenVPN. It then goes through 4 steps:

① The packet is copied in the enclave.

② It is processed by one or more middlebox functions according to the configuration (using Click). Depending on the specific function, the packet headers or its payload may be modified. The whole packet can also be marked for rejection (*e.g.*, due to firewall or IDPS rules).

③ According to the result of the middlebox functions, the packet is forwarded to the next step, or discarded.

④ Finally, the packet is signed and encrypted and then copied outside of the enclave, where it is passed back to the VPN client running in untrusted space, for transmission over the physical network.

Each VPN-related packet arriving from the network is processed slightly differently:

① The packet is copied in the enclave, its signature is checked and its contents are decrypted.

②, ③ *ditto*

④ The packet is passed back to the VPN client running in untrusted space, for transmission over the *virtual network interface*.

7.4.4. Attestation and key management

To achieve the desired level of security (*R2*), EndBox leverages the attestation facilities offered by Intel SGX [8]. In EndBox, keys are managed using a certificate authority (CA) operated by network owners. The public key of the CA is hardcoded in enclave binaries as part of the compilation to prevent MitM attacks.

Figure 7.8 shows the steps that EndBox applies in order to attest the correctness of the enclave, as well as protect and sign VPN keys.

① An asymmetric key pair is generated in the enclave; the private key never leaves the enclave.

② The EndBox client creates a *report* containing the public key and passes it to the quoting enclave (QE) to obtain a *quote*.

③ The quote is forwarded to the CA.

④ In turn, the CA relays the quote to IAS.

⑤ Provided that IAS replies positively, and that the quote contains a known *measurement*, the CA signs the public key, hence creating a certificate. It also generates a shared symmetric encryption key, which will be used to decrypt configuration files.

⑥ The certificate, along with the symmetric key—which is encrypted with the enclave's public key—are provisioned to the enclave.

⑦ The enclave checks that the received certificate is genuine, *i.e.*, signed by the CA. It then stores the key-pair, the certificate, as well as the symmetric key to disk using the *sealing* feature provided by SGX.

The client can now use that certificate to connect to the VPN server. As all keys are *sealed* to disk, the EndBox client can start immediately without having to perform the attestation procedure on future starts (provided the system configuration of the machine stays the same).

7.4.5. Processing encrypted network traffic

Many middlebox functions, *e.g.*, DPI and caching, operate on the payload of packets. Nowadays, a large part of the traffic is encrypted [73], which means that the payload appears as an opaque blob to middleboxes. There are different state-of-the-art solutions to tackle this problem: (i) middleboxes perform the equivalent of an MitM attack on users; (ii) modify the TLS protocol in a way that allows middleboxes to intercept traffic [135, 134], and (iii) searchable or homomorphic encryption schemes [157]. Each of them have severe disadvantages: they break end-to-end security, are incompatible with existing technologies, are impractical or are notoriously slow.

Therefore, we devise a new approach to access encrypted network traffic. The idea is that client applications, *e.g.*, web browsers, will transparently share their symmetric encryption keys with EndBox. The application will be linked against a modified TLS library, which will forward all negotiated session keys to the trusted Click instance running inside the EndBox client. A new Click element can decrypt the packets inside the middlebox pipeline.

The client does not need to trust non-standard CAs, as it receives the real certificates offered by the services it accesses. Moreover, we keep the TLS protocol as-is. Our approach to analyze encrypted traffic also works with TLS version 1.3, which existing middleboxes could not handle correctly [151].

Note that transferring keys to the EndBox enclave is not a security risk for the client: the keys are generated by the TLS library running in *untrusted* space (*i.e.*, not secured by SGX). Copying them to the more secure EPC therefore does not degrade the overall security in the slightest.

Regarding the notion of analyzing encrypted traffic in the first place, we think that our solution is applicable to our targeted scenarios. In an enterprise network, employees need to trust their employer to some extent and should refrain from handling private matters using company networks. In the ISP scenario, we assume that customers opt-in for traffic analysis to improve security; they explicitly consent.

7.5. Implementation

The implementation of EndBox is based on OpenVPN 2.4.0 [52], the latest version of the Click software router [108], and the TaLoS library for terminating TLS connections inside SGX enclaves [12]. We use OpenVPN as the basis for the EndBox client because it is open-source, has relatively few dependencies, and is implemented in user-space. This allows us to port parts of its implementation to an SGX enclave. TaLoS is based on LibreSSL and acts as a drop-in replacement running in SGX enclaves for existing applications.

We use the Intel SGX SDK 1.9 [84] to define *ecalls* and *ocalls* as well as to handle the life cycle of the enclave. In addition, we use the trusted—but functionally limited—C library implementation shipped with the SDK. We extend it further with functions used by OpenVPN and Click. The EndBox implementation also accesses trusted time through the SDK in order to implement traffic shaping.

EndBox relies on Click to implement middlebox functions. To configure it, so-called *elements* are interconnected to form a pipeline. An element receives packets, processes them and then forwards them to other elements. We choose Click because it is widely used, and is easily extensible. Moreover, its configuration can be hot-swapped. Click ships many elements that implement various middlebox functions out-of-the-box; we use them to implement EndBox's own middlebox functions. We also extend Click by adding custom elements that provide (i) an IDPS; (ii) decryption of application-level traffic, and (iii) traffic shaping using the trusted time source provided by SGX.

7.5.1. Optimizations

We implement several optimizations to improve the performance and security of EndBox. We describe those in this section and evaluate them further in section 7.6.2.

Enclave transitions

The performance of SGX enclaves is negatively impacted by transitions between trusted and untrusted code (*cf.*, section 4.3.4). To reduce this cost, we place parts of the encryption logic of OpenVPN into the enclave to reduce the number of enclave transitions per processed packet to a single *ecall* per packet.

Client-to-client communication

When an ENDBOX client communicates with another ENDBOX client on the same virtual network, packets would end up being processed by both ENDBOX instances. As both clients are trusted and enforce the same configuration, processing packets twice is wasteful. To prevent this waste, ENDBOX clients flag outgoing packets after they have been processed by Click. The flagging mechanism works by setting the differentiated services (DS) field [15] of the Internet protocol (IP) header to `0xeb`. In order to prevent external attackers from sending packets that contain this value, the ENDBOX server erases the field from incoming packets when it is set to `0xeb`. As all packets are integrity-protected by OpenVPN, flagged packets cannot be forged within the VPN.

7.5.2. Secure enclave interface

The enclave interface of the ENDBOX client consists of 90 calls: 70 *ecalls* and 20 *ocalls*. Most of the *ecalls* are only used during the initialization of OpenVPN and Click. Four *ecalls* are executed during normal operation: (i) packet encryption; (ii) packet decryption; (iii) message authentication code (MAC) generation, and (iv) MAC verification. While *ecalls* (i) and (ii) are triggered by normal traffic, *ecalls* (iii) and (iv) are used to protect the integrity of the control channel of OpenVPN.

With the exception of the ENDBOX-specific en- and decryption and Click initialization *ecalls*, all the other *ecalls* match those of TaLoS [12], which perform security checks. The *ocalls*, on the other hand, perform different tasks, such as managing untrusted memory and accessing encrypted configuration files. Note that they could be completely removed by using in-enclave configuration files and *exitless* enclave services [140].

To ensure that the interface is secure, we closely examined all *ecalls* and *ocalls* and augmented them with sanity checks on arguments and return values. We also check the bounds of pointers to guarantee that they point to enclave memory.

7.6. Evaluation

We evaluate the security and performance of ENDBOX by discussing different attacks on ENDBOX and performing different measurements.

7.6.1. Security evaluation

We discuss potential attacks against ENDBOX and state how it can defend against these or why they are not applicable.

Bypassing middlebox functions

A malicious client may try to access the network without using EndBox. In this case, the network needs to be guarded by a stateless firewall that only allows EndBox-generated traffic. It therefore becomes impossible to bypass the middlebox functions as traffic will be dropped by the firewall.

Using old or invalid middlebox configurations

An attacker may rollback configuration updates, or use unauthorized configurations. Once an adjustable grace period for an update has passed, the server only accepts EndBox clients that use the currently valid configuration. The EndBox client and server periodically exchange ping messages containing configuration information to prevent lawful clients from using stale configurations.

Replaying traffic

Traffic replay attacks are thwarted by the replay protection offered by OpenVPN.

Denial-of-service attacks

Malicious clients can prevent enclaves from starting or being entered, as the enclave life cycle is managed by untrusted code. However, this would result in the inability of the client to communicate with the network. On the other hand, a DoS attack on the EndBox server would have the same effect as on a traditional centralized middlebox deployment. Existing techniques exist to mitigate this kind of attacks [54].

Downgrade attacks

Attackers could try to force the usage of a weaker TLS version or cipher. OpenVPN implements server-side checks that ensure that a minimal TLS version is used. On the client-side, the corresponding check happens within the enclave during connection establishment and therefore cannot be circumvented.

Interface attacks

A client may try to break into the enclave by manipulating the parameters at the enclave interface similar to Iago attacks [34]. To mitigate such attacks, every *ecall* and *ocall* has been augmented with checks on input parameters and return values (*cf.*, section 7.5.2). In addition, EndBox exposes a limited interface with a restricted attack surface.

Failure of a middlebox

If a middlebox fails, only the client running this middlebox is impacted; other clients and the managed network remain unaffected. This differs from the behavior of traditional centralized middlebox setups in which a failure would affect many clients or even whole networks. In contrast, the failure of the ENDBOX server managing all VPN connections is equivalent to a failure of traditional centralized middleboxes, resulting in network outages.

7.6.2. Performance evaluation of different middlebox functions

We evaluate the performance of ENDBOX across 7 machines of two different sets of specifications. Class $\boxed{\text{A}}$ consists of 5 machines equipped with SGX-capable 4-core Intel Xeon v5 CPUs with 32 GB of memory, while the other class $\boxed{\text{B}}$ machines consist of 2 machines with non-SGX-capable 4-core Xeon v2 CPUs and 16 GB of memory. All machines are configured with Intel *HyperThreading* feature turned on. They are all connected to a 10 Gbit/s switched network. The network is configured to support 9000 B jumbo frames.

We conduct throughput measurements using the *iperf 3* [116] tool, while we use Internet control message protocol (ICMP) echoes to measure latency. We report average values after 10 consecutive runs.

Our performance evaluation is conducted on a few different setups that are designed to highlight where ENDBOX itself affects performance and where the change in performance is due to the components that it uses. We refer to those different setups as such.

Vanilla OpenVPN Unmodified OpenVPN 2.4.0.

OpenVPN + Click OpenVPN with additional server-side processing by Click.

ENDBOX in simulation mode (sim) Our contribution running in *insecure* simulated SGX mode to showcase the overhead of partitioning the VPN client.

ENDBOX in hardware mode (SGX) Our contribution running in protected mode.

We implement 4 representative middlebox functions to showcase in our evaluation (plus a baseline). Each function is based on one or more Click element. Some elements are original, while some come standard with Click.

Forwarding (NOP) The first element we consider provides a baseline for our measurements. It forwards packets without accessing or modifying any headers.

Load balancing (LB) The `RoundRobinSwitch` element shipped with Click balances IP packets or TCP flows across several machines.

IP firewall (FW) The `IPFilter` element shipped with Click implements a firewall function. It accesses packet headers and controls traffic based on a set of rules. For our evaluation, we use a set of 16 rules that do not match any packet, forcing the firewall to go through them all for all packets.

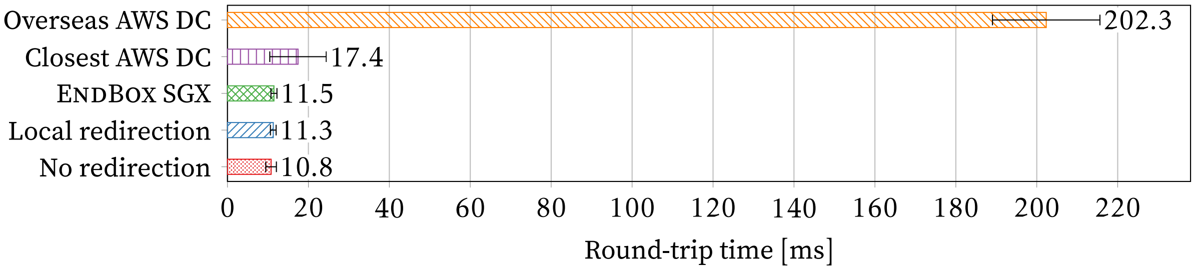

Figure 7.9.: Average ICMP echo RTT for different redirection methods.

Intrusion detection and prevention system (IDPS) Our IDPS is based on a custom Click element called `IDSMatcher`. It monitors network traffic for unauthorized accesses and policy violations. We support Snort [152] rule sets; we execute its string matching algorithm [3] using a library that comes from the Snap framework [162]. For the evaluation, we extract a subset of 377 rules from the Snort community rule set. Akin to the firewall, the rules do not match any of the packets that we generate during the evaluation.

Distributed denial-of-service prevention (DDoS) DDoS attacks can generally be mitigated by throttling or dropping packets that occur repeatedly, or by detecting and dropping packets with a spoofed source address. We implement this middlebox function by rate-limiting identical packets using two custom Click elements, `IDSMatcher` as presented above, and `TrustedSplitter`. The latter shapes traffic to a given capacity in a trusted way. As obtaining time from the trusted source offered by SGX requires expensive calls, the `TrustedSplitter` element samples timestamps by issuing calls after a certain configurable number of packets has been processed. We set this number to 500 000 for our measurements. In the *OpenVPN+Click* setup, we use a similar Click element called `UntrustedSplitter` which instead obtains timestamps using system calls.

Latency of different middlebox deployment strategies

We evaluate the impact on latency incurred by EndBox, as this has a notable influence on user experience. We use the *forwarding* middlebox function (NOP) and perform local experiments using $\boxed{\text{A}}$ machines. For cloud-based measurements, we rely on Amazon Web Services (AWS) Elastic Compute Cloud (EC2) and use `m3.medium` instances with 1 virtual CPU and 4 GB of RAM in two different data centers (DCs). We create a setup of software middleboxes executing in EC2 and measure the round-trip time (RTT) of ICMP echoes from a fixed location.

Figure 7.9 shows the average RTT for different redirection methods: (i) unaltered setup with no redirection, middlebox or VPN; (ii) local redirection through a VPN and a server-side middlebox using OpenVPN+Click; (iii) redirection through EndBox running in SGX, and (iv) and (v) redirection through OpenVPN+Click middleboxes deployed on EC2 instances in two different AWS regions.

The results show that the latency overhead induced by a cloud-based middlebox deployment greatly depends on the location of the DC of the cloud provider. Nevertheless, even using the

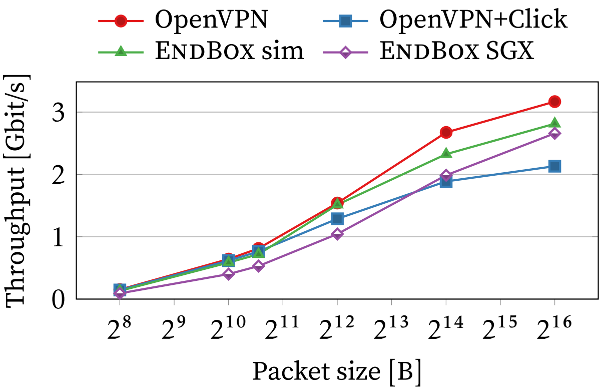

Figure 7.10.: Average throughput of different setups for various packet sizes.

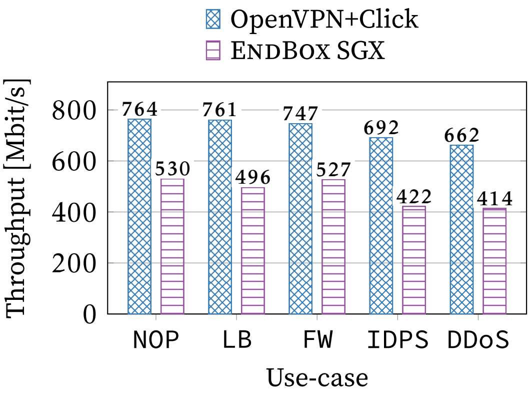

Figure 7.11.: Average throughput for different use-cases.

closest DC offered by AWS incurs a latency overhead of 61 %; a figure which rises to 1773 % in the case of an overseas DC. Notably, EndBox, only increases the latency by 6 %, showing the viability of the deployment strategy we chose.

Optimizations

We evaluate the positive impact of the optimizations described in section 7.5.1. Reducing the number of enclave transitions per packet results in a substantial 342 % increase in throughput. However, optimizing client-to-client communication has no effect on throughput, but decreases latency between clients by up to 13 % for the IDPS use-case.

Throughput

In this experiment, we measure the maximum throughput reached by each of our four different setups. We generate packets of sizes ranging from 256 B to 64 KiB. The maximum transmission unit (MTU) of our network is set to 9000 B.

The results are shown in figure 7.10. As expected, higher throughput is measured with larger packets as the per-packet fixed costs are better amortized. Chiefly, we see that the performance overhead of EndBox ranges from 16 % for large packets to 39 % for small packets. Larger packets permit higher throughput as they cause less enclave transitions.

In any case, we see that EndBox is well-optimized as the performance overhead of a regular server-side OpenVPN+Click instance (EndBox uses both pieces of software internally) ranges from 5 % to 29 %. Finally, we observe that OpenVPN+Click achieves a throughput almost one third lower than vanilla OpenVPN for large packets due to Click's packet fetching.

In the next experiment, we evaluate the throughput that we can reach when each of our five representative middlebox functions are active. We compare EndBox against an equivalent server-side OpenVPN+Click setup. Figure 7.11 shows the average throughput measured between a class $\boxed{A}$ client machine and a class $\boxed{B}$ server, using a standard packet size of 1500 B. NOP represents a baseline.

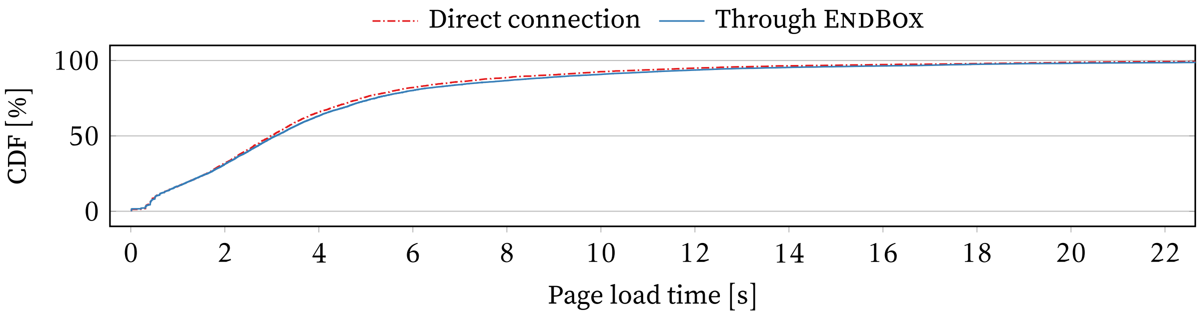

Figure 7.12.: Load time CDF of 1000 popular websites, with and without EndBox.

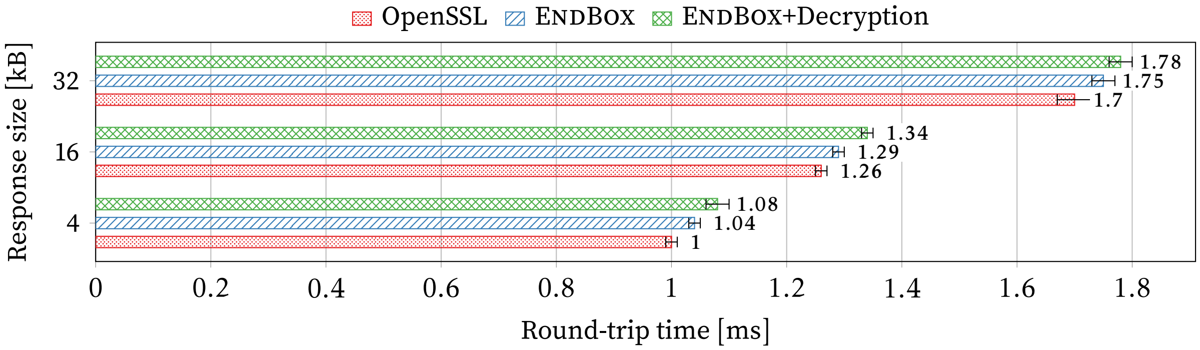

Figure 7.13.: Latency impact of the TLS decryption feature of EndBox.

First, we observe that the biggest performance impact in the case of OpenVPN+Click is the DDoS use-case, with an overhead of 13 %, which is rather low. In comparison, EndBox incurs an additional overhead of *circa* 30 % for the NOP, LB, and FW use-cases. The more computationally intensive use-cases IDPS and DDoS show an overhead of 39 %. As explained above, the overhead is lower for larger packets.

User-centric latency measurement

In this experiment, we establish the actual impact that a user sees when they use EndBox. We load the homepage of each of the 1000 most popular websites as listed by Alexa [5]. We use the PhantomJS [78] headless web browser so as to measure actual user-facing page load times (*i.e.*, including images, stylesheets, *etc.*). We run the experiment with direct network access and through EndBox. The results are depicted in figure 7.12. We see that it takes approximately the same time to load those web pages with or without EndBox. Thus, we can conclude that the user-facing latency overhead of EndBox is negligible.

Handling encrypted traffic

As mentioned in section 7.4.5, EndBox is able to transparently and securely decrypt TLS traffic. We measure the overhead of this feature by fetching static web pages of different sizes using HTTPS GET requests, *i.e.*, hypertext transfer protocol (HTTP) through TLS. We use

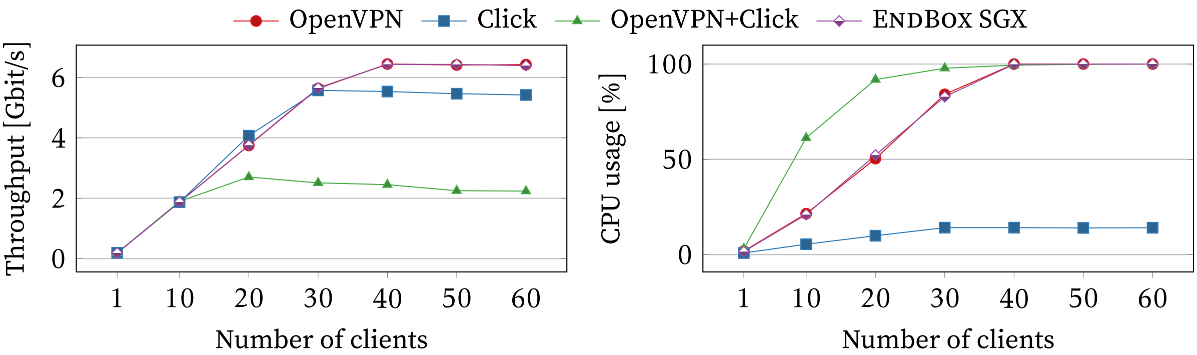

Figure 7.14.: Scalability of EndBox and its constituting components.

three setups, (i) using the unmodified OpenSSL library (the baseline); (ii) using our modified OpenSSL library through EndBox but without performing any packet decryption, and (iii) same as (ii), but packets are decrypted inside the enclave.

The results are shown in figure 7.13. We see that the overhead introduced by our custom OpenSSL library (which forwards symmetric keys to the enclave) along with packet decryption is under 8 %.

Scalability

After evaluating user-facing properties of EndBox, we evaluate its scalability, which is an important factor to network operators. The experiment consists in measuring the throughput and CPU usage on the server-side. Throughput is aggregated over all virtual interfaces set up by OpenVPN server instances, *i.e.*, one per client as OpenVPN is single-threaded. CPU usage is measured across all cores, *i.e.*, 100 % represents all cores being fully utilized. We run multiple EndBox client on 5 class $\boxed{A}$ machines. Each client generates a fixed 200 Mbit/s of traffic using *iperf3*. One class $\boxed{B}$ machine runs the EndBox server, while another serves as the *iperf3* endpoint.

Scalability of individual components First, we evaluate the scalability of the various components that EndBox is made of. Using the forwarding middlebox function (NOP), we see in figure 7.14 that OpenVPN and EndBox saturate at the same throughput of 6.5 Gbit/s, with an almost identical CPU usage. This shows that executing middlebox function on the client side has no impact on throughput or CPU usage on the server side. Maximum per-client throughput is achieved with 40 clients per server. Our investigations show that the bottleneck is the cryptography performed by OpenVPN.

Click requires a substantial amount of CPU cycles, which become the bottleneck as the CPU becomes fully utilized earlier than with EndBox. The throughput of Click alone is limited to 5.5 Gbit/s as its process cannot handle more packets. Finally, we see that the throughput achieved by OpenVPN+Click is only 2.5 Gbit/s, as the full load of all clients is processed centrally by the server CPU.

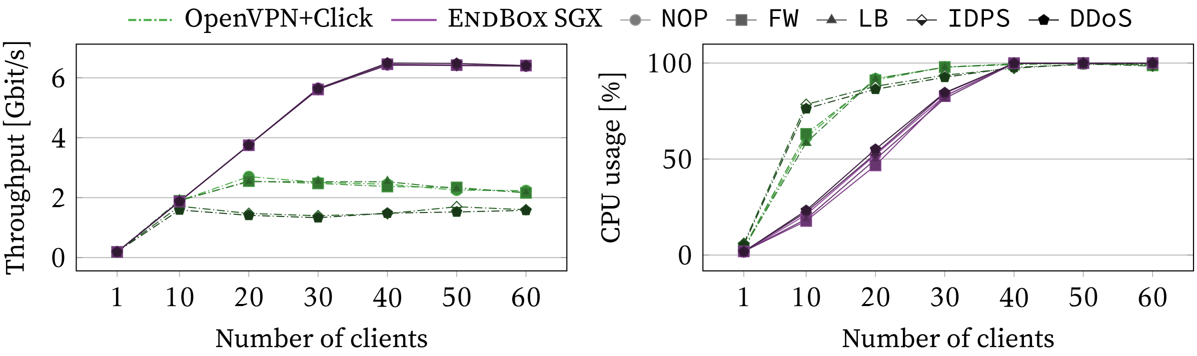

Figure 7.15.: Scalability of ENDBOX and an equivalent centralized middlebox in different use-cases.

Scalability of different use-cases Now, we repeat the same measurement for our five representative use-cases. The previous results for OpenVPN+Click and ENDBOX form our baselines. The goal of the experiment is to show how ENDBOX scales with the number of clients when different middlebox configurations are applied.

Figure 7.15 shows the results. As previously mentioned, the VPN server saturates at 40 clients and becomes the bottleneck of ENDBOX at a maximum throughput of 6.5 Gbit/s for all use-cases. On the other hand, OpenVPN+Click executes the middlebox functions centrally on the server itself, and reaches this limit with 30 clients at a maximum throughput of only 2.5 Gbit/s. The computationally-intensive IDPS and DDoS middlebox functions only achieve a maximum aggregate throughput of 1.7 Gbit/s. Pattern matching on network packets is what overloads the central middlebox.

To conclude the scalability evaluation of ENDBOX, we report that it scales linearly with the number of clients. For 60 clients, ENDBOX achieves a 2.6× higher throughput across all use-cases than a comparable fully-centralized middlebox, and 3.8× higher for computationally-intensive workloads. Thus, we show that ENDBOX performs especially well for CPU intensive middlebox functions.

7.7. Summary

In this chapter, we presented ENDBOX, a scalable system that enables the secure deployment and execution of middlebox functions on untrusted client machines thanks to Intel SGX. It scales linearly with the number of clients, thereby achieving 2.6× to 3.8× higher throughput than a traditional deployment at the core of a managed network. Despite being distributed, configuration changes to ENDBOX-based middlebox services are centrally controlled and enforced. Finally, encrypted application traffic can be efficiently and securely decrypted and filtered using ENDBOX, due to its location at the client side.

Part III.

In-network line-speed data processing

Chapter 8.

In-network compression at line speed

8.1. Introduction

The steady increase in network traffic [95] is a challenge for network operators. A potential way to decrease the size of individual network packets would be to compress them. However, compression is usually a computationally-intensive operation that correspondingly requires considerable resources.

In this chapter, we present **ZipLine**, the first in-network data (de)compressor operating at line-speed. ZipLine leverages a novel data deduplication technique: generalized deduplication (GD) [177]. GD offers compression levels that can outperform traditional schemes at a lower memory footprint [184].

We revise the processing workflow of GD to fit our target, the Intel Tofino programmable switch (*cf.*, section 3.2). Thanks to its hardware hashing units, we can implement the algorithm in a single pass, providing latency-free compression at line-rate simultaneously on all ports of the appliance.

Our work highlights that in-network *processing* is practical, as elaborate algorithms can indeed be implemented on programmable switches. We report on several implementation caveats and the lessons learned while doing so.

8.2. Background

This section provides an overview of the compression algorithm behind ZipLine and the coding techniques used for its implementation.

8.2.1. Generalized deduplication

GD is a generalization of the concept of data deduplication [177], where the system first applies a transformation function on a data chunk to split it into a pair of values: a *basis* and a *deviation*. Then, the system proceeds to deduplicate that basis against previously received ones. All deviations are kept to be able to invert the process.

To illustrate how GD operates, let us consider the following example. We feed the following 7-bit sequences into a transformation function based on Hamming codes (*cf.*, section 8.2.3).

$$\{\texttt{0000000}, \texttt{0000001}, \texttt{0000010}, \texttt{0000100}, \texttt{0001000}, \texttt{0010000}, \texttt{0100000}, \texttt{1000000}\}$$
$$\{\texttt{1111111}, \texttt{1111110}, \texttt{1111101}, \texttt{1111011}, \texttt{1110111}, \texttt{1101111}, \texttt{1011111}, \texttt{0111111}\}$$

For all chunks on the first line, we get a single 4-bit basis `0000` and all eight possible 3-bit deviations that indicate what bit to flip (if any). Similarly, the chunks on the second line all give out the basis `1111`, with the same deviations as for the first line.

By using a registry containing the most commonly used bases instead of the original data chunks, we are able to compress data. Let us consider the following 42-bit sequence made of six different 7-bit data chunks.

`0000000 1111111 0100000 1111011 1000000 1011111`

As previously shown, the individual 7-bit chunks map to only two bases. We can uniquely identify each basis using a 1-bit identifier. Thus, the data can be represented with a dictionary containing the two bases (2 × 4 bits) and a sequence of 1-bit basis identifiers and 3-bit deviations, for a total of 24 bits transmitted.

`0 000  1 000  0 111  1 100  0 101  1 110`

In general, using longer sequences allows more chunks to map to the same basis. The result is that thousands or even millions of chunks can be mapped to the same basis, increasing the potential for compression.

In this work, we consider Hamming codes as the core component of the transformation function for GD, as did Vestergaard *et al.* [177] and Göttel *et al.* [63]. The main motivation is that they can be implemented using equivalent cyclic redundancy checks (CRCs) which can be computed in hardware on programmable switches. We show this equivalence in sections 8.2.2 and 8.2.3.

8.2.2. Cyclic redundancy check

Let us consider a block of data B with n bits, where B can be expressed as a polynomial $B(x) = b_0x^0 + b_1x^1 + \ldots + b_{n-1}x^{n-1}$ where b_i is the i-th bit of B. We consider b_{n-1} as the most significant bit (MSB) of B. Computing an m-bit long CRC requires computing the long division of $B(x)$ with a generator polynomial $g(x) = g_0x^0 + g_1x^1 + \ldots + g_mx^m$. The residue of this division is the CRC value.

One of the properties of CRCs is that $\text{CRC}(A \oplus B) = \text{CRC}(A) \oplus \text{CRC}(B)$. Hence, pre-computing all n-bit sequences that have a single bit set allows us to compute any sequence of n bits by XORing the appropriate CRCs sequences.

$$\begin{aligned}\text{CRC}(B) = b_{n-1}\text{CRC}(\texttt{10}\ldots\texttt{00}) \oplus b_{n-2}\text{CRC}(\texttt{01}\ldots\texttt{00}) \oplus \ldots \\ \oplus\, b_1\text{CRC}(\texttt{00}\ldots\texttt{10}) \oplus b_0\text{CRC}(\texttt{00}\ldots\texttt{01})\end{aligned}$$

This computation can be represented in matrix form as $\text{CRC}(B) = BH^T$, where H is a *parity-check matrix*.

Table 8.1.: Hamming code $(7,4)$ and CRC-3 equivalence.

	Hamming $(7,4)$		CRC-3	
Input	Error	Syndrome	Polynomial	CRC-3
(0000001)	0	(001)	x^0	(001)
(0000010)	1	(010)	x^1	(010)
(0000100)	2	(100)	x^2	(100)
(0001000)	3	(011)	x^3	(011)
(0010000)	4	(110)	x^4	(110)
(0100000)	5	(111)	x^5	(111)
(1000000)	6	(101)	x^6	(101)

8.2.3. Hamming codes

Hamming codes are block codes that convert k-bit messages into n-bit messages by adding m parity bits [26]. More specifically, $n = 2^m - 1$ bits and $k = n - m = 2^m - m - 1$ bits. They are generated using a generator matrix G as follows:

$$G = \begin{bmatrix} g_{0,0} & g_{0,1} & \cdots & g_{0,n-1} \\ g_{1,0} & g_{1,1} & \cdots & g_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ g_{k-1,0} & g_{k-1,1} & \cdots & g_{k-1,n-1} \end{bmatrix}$$

where the size of the matrix is $k \times n$.

Such a code can be transformed to *systematic* form, where the original message is directly embedded in the codeword and the parity bits are clearly separated from it as $G = \begin{bmatrix} I_k & P \end{bmatrix}$, which is achieved by performing row operations to eliminate components. I_k indicates an identity matrix of size $k \times k$.

For our work, we consider a case where the order of parity and identity matrices are shifted, *i.e.*, $G_s = \begin{bmatrix} P & I_k \end{bmatrix}$, as it matches the output of CRC functions. The *parity-check matrix* H of size $m \times n$ is given by $H = \begin{bmatrix} I_{n-k} & P^T \end{bmatrix}$, which is actually the same as for computing the CRC with the same generator polynomial [26]. A message u of size $1 \times k$ is encoded into the codeword c of size $1 \times n$ by $c = u\,G_s$.

For decoding, the received sequence is $B = c + e$, where e is the error pattern. An all-zero e means there are no errors. The matrix H can be used to detect whether any errors occurred by calculating the *syndrome* vector $s = BH^T = (c + e)\,H^T = eH^T$, due to the fact that $G_s H^T = 0$ (by construction of H and G_s). We can construct a lookup table that maps the different 1-bit error patterns to corresponding s values.

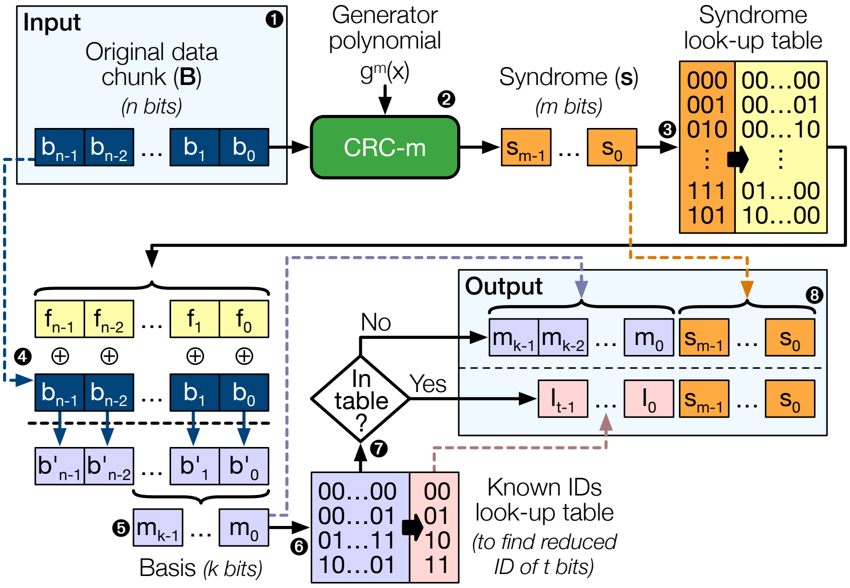

Figure 8.1.: GD encoding workflow on Tofino.

8.2.4. Connection of Hamming codes to CRCs

A CRC-m using a generator polynomial that is suitable to a Hamming code (n,k) can help compute the syndrome and parity bits in the decoding and encoding steps. More specifically, the computed CRC-m value will be equivalent to the syndrome computed for a $(n,k) = (2^m - 1, 2^m - m - 1)$ Hamming code as long as (i) the generator polynomial for a Hamming code is used as parameter for the CRC-m generator polynomial, and (ii) the size of the input data to be computed by the CRC-m is $n = 2^m - 1$ bits as expected by a standard Hamming code. Table 8.1 shows an example of the equivalence between the $(7,4)$ Hamming code and CRC-3 with a carefully chosen generator polynomial.

8.3. Approach

Our approach uses the GD algorithm presented in section 8.2.1 to implement compression of network packets in a way that is implementable on a readily-available programmable switch. The encoding workflow is shown in figure 8.1. Let us consider a data payload B of size n, part of an incoming network packet ❶. The first action to perform is to compute the syndrome s using a Hamming decoder (mapped to an equivalent CRC) ❷. The syndrome tells us which bit in the original data needs to be flipped. We achieve this by having a table ❸ that matches to the correct mask f to be XORed to the original data B ❹ according to s. The result of this XOR operation b' is truncated to the rightmost k bits to form a basis $(m_{k-1}, \dots, m_0)$ ❺. As several data chunks share the same basis, we take this opportunity to replace them with shorter identifiers I should the basis been seen before ❻,❼. In order for the recipient to be able to find B again, we attach the syndrome s to the compressed packet ❽.

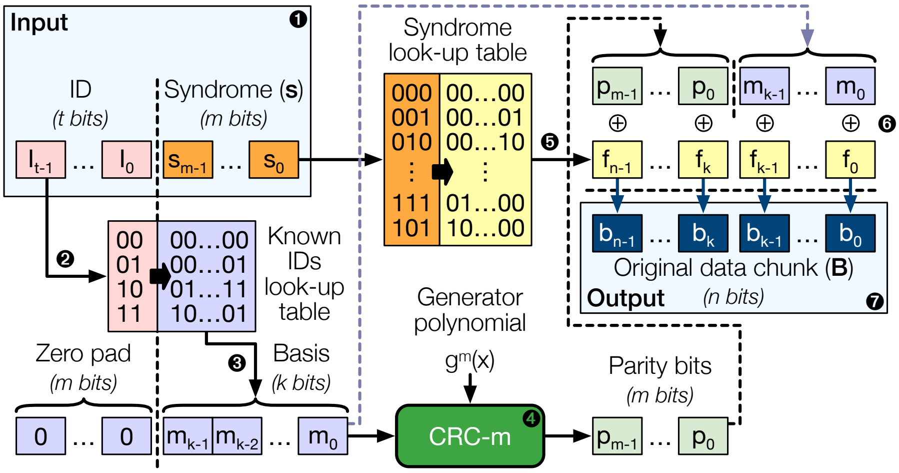

Figure 8.2.: GD decoding workflow on Tofino.

The decoding workflow is shown in figure 8.2. We consider that a packet containing a compressed basis has arrived ❶. For an uncompressed basis, the workflow instead starts at step ❸. On the recipient's side, there must exist a table that maps short identifiers I to bases, that is synchronized with its counterpart in the encoder ❷. We zero-pad the recovered basis ❸ and then feed it to the same CRC generator as on the encoder ❹ to get p, hence restoring the bits that we truncated in the encoder. In parallel, we use the same syndrome look-up table as in the encoder to identify which bit needs to be flipped according to s. This gives us the appropriate mask f to be XORed ❺. We apply the mask to flip the right bit out of the concatenation of p and the basis ❻ to successfully restore the original data B ❼.

8.4. Related work

Offloading heavy-duty processing from end-node central processing units (CPUs) to the network itself has been studied in several research efforts.

NetEC [148] implements efficient in-network erasure codes (ECs). It aggregates data streams using Reed-Solomon [149] so that they leave the switch in aggregated fault-tolerant form.

P4-enabled switches can perform different types of aggregations, *e.g.*, over several header frames for fixed-size [181] or variable-size [180] payloads, while achieving line-rate throughput. Small payloads are stored in *registers* of the switch and are recirculated via the ingress parser until the maximum transmission unit (MTU) is reached, at which point the generated larger packet is released.

Chen [35] presents a non-trivial in-network design and implementation of advanced encryption standard (AES) for different payload sizes. Packets go through the switch pipeline multiple times to complete the ten rounds used for AES-128. In contrast, ZipLine does not need to recirculate packets as GD can be implemented in a single round.

Although there exist switches that perform on-the-fly compression of data streams [32, 183, 76], these operate at layers 3 and above. Our approach can support a wider set of protocols by operating at layer 2. Also, thanks to P4-compatible application-specific integrated circuits (ASICs), ZipLine can operate on all ports of a switch at line-rate simultaneously. ZipLine will also automatically improve in the future whenever a more-powerful P4-compatible ASIC gets on the market.

8.5. Implementation

Our implementation uses the capabilities provided by the Intel Tofino programmable switch. It is implemented in $P4_{16}$ with Tofino-specific proprietary extensions, *i.e.*, Tofino Native Architecture (TNA), using Capilano software development environment (SDE) version 9.2.0. The control plane is implemented using Python and Barefoot runtime (BfRt).

We favor simplicity whenever possible. In this mindset, we choose to develop everything from scratch, which also reduces the likelihood of resource allocation issues. We settle on Ethernet-based framing to provide compatibility with regular Ethernet network cards and guarantee inter-connectivity.

We define three different types of packets: (i) regular, yet unprocessed packets; (ii) processed, but uncompressed packets, comprised of a *basis* and a *syndrome*, and (iii) processed and compressed packets, which replace the *basis* with a shorter identifier. Packet type (i) can be any Ethernet packet that arrives in the switch. Our implementation takes such packets as input, and then transforms them into types (ii) or (iii). The latter type can only be produced when there exists a basis-identifier mapping for the basis that the original type (i) packet maps to. Such mappings are initialized once a yet-unseen basis is computed. They remain valid for as long as possible. A least recently used (LRU) cache eviction policy decides when to recycle an identifier to map to a different basis. More details about this stateful part of our implementation are provided in later paragraphs.

The foundation of our implementation is the GD algorithm as described in section 8.2.1. Notably, GD uses Hamming codes to work as they are equivalent to some particular CRCs. TNA offers a native component to compute CRCs, as such codes are commonly found in many network protocols. We extensively rely on this component to efficiently implement the key steps of the GD algorithm on Tofino, namely the computation of *syndromes*. Table 8.2 shows generator polynomials for Hamming codes and the expected input parameter for the CRC-m module of the Tofino chipset.

The next part of the workflow requires flipping one bit in the input data according to the syndrome that we obtained in the previous step. We use a P4 table with constant entries that are pre-computed using a short C++ program making use of Boost CRC library [179]. The entry that matches the syndrome is XORed to the data, hence flipping the appropriate bit of the sequence. This transformation creates the *basis*.

In order to make it possible to compress packets, we need to replace *syndrome* + *basis* couples with shorter identifiers (IDs). One possible way to select an identifier for a given basis would

Table 8.2.: Generator polynomials for Hamming codes and parameters for a CRC-m.

Code	Generator polynomial	CRC-m parameter
$(7,4)$	x^3+x+1	`0x3`
$(15,11)$	x^4+x+1	`0x3`
$(31,26)$	x^5+x^2+1	`0x05`
$(31,26)$	$x^5+x^4+x^2+x+1$	`0x17`
$(63,57)$	x^6+x+1	`0x03`
$(127,120)$	x^7+x^3+1	`0x09`
$(255,247)$	$x^8+x^4+x^3+x^2+1$	`0x1D`
$(511,502)$	x^9+x^4+1	`0x00D`
$(511,502)$	$x^9+x^8+x^7+x^6+x^5+x+1$	`0x0F3`
$(1023,1013)$	$x^{10}+x^3+1$	`0x009`
$(2047,2036)$	$x^{11}+x^2+1$	`0x005`
$(4095,4083)$	$x^{12}+x^6+x^4+x+1$	`0x053`
$(8191,8178)$	$x^{13}+x^4+x^3+x+1$	`0x01B`
$(16383,16369)$	$x^{14}+x^8+x^6+x+1$	`0x143`
$(32767,32752)$	$x^{15}+x+1$	`0x003`

be to use cryptographic hashes, however those cannot be computed in one pass on our programmable switch. The alternative is therefore to use arbitrary IDs. We involve the control plane to manage the pool of identifiers. Unknown bases are sent up by the data plane to the control plane by means of *digests*, as provided by TNA. Recording a new basis-ID mapping is done in two phases: first, the control plane chooses an identifier to assign to the basis. When there are unused identifiers, the control plane selects the least recently used one. Should all identifiers be in use, an LRU policy is applied to evict and recycle an identifier to accommodate the most recent basis. Setting a time to live (TTL) that is automatically decreased as time elapses on a particular table entry is a feature that is provided by TNA. With the identifier now selected, the control plane first sets the reverse mapping (ID-basis) in the destination switch to make sure that compressed packets can always be uncompressed. The control plane can finally add a corresponding entry in the source switch, to map the newly-discovered basis to the chosen identifier. Subsequent packets sharing the same basis are then transmitted in a shorter *syndrome + identifier* form.

Last, we add *counters* to our program to provide easily-accessible statistics of the inner workings of ZipLine. In particular, packets are classified according to how they are transformed (*e.g.*, raw packet to syndrome + basis, syndrome + identifier to raw packet, *etc.*).

8.6. Lessons learned

We report on several lessons that we learned during the development process of ZipLine. In particular, we highlight some potential intricacies that we encountered when developing

ZipLine with P4$_{16}$/TNA for the Intel Tofino platform. P4 is a flexible language that is mainly designed to efficiently process network packet headers. Similarly, the hardware design of the Tofino chip is tailored to the same goals. We use those pieces of technology in quite an unorthodox way that largely differs from traditional packet routing. As a consequence, we had to adapt our implementation to circumvent some technical limitations.

Header declarations in P4$_{16}$ must be aligned on byte boundaries. However, the Hamming codes that we use are never aligned on bytes (*cf.*, table 8.2). While the P4 language is well-suited to bit-level granularity, the hardware and its associated compiler are optimized for byte-aligned data. A side-effect is that we have to introduce padding bits in our program to cope with non-byte-aligned sizes for the program to successfully compile. In turn, this introduces a loss of goodput, limiting the range of interesting parameter values to only a few.

In our original design, we placed as much of the code as possible in the data plane, *i.e.*, in the P4 program. This was made possible by leveraging *registers*, *i.e.*, user-accessible memory in the Tofino data plane pipeline. Doing so allowed us to achieve line-rate performance and perceptibly instantaneous learning of new basis-ID pairs. However, as every piece of code running in the data plane must be able to execute in constant time, many algorithms are out-of-reach, *e.g.*, algorithms that need a complete view over an array of registers. Therefore, we settled on storing basis-ID pairs in regular match-action tables and manage them with the control plane, which is implemented in a regular programming language (*i.e.*, Python in our case). This allows us to use features such as *digests* and *table entry-specific TTLs* that the TNA framework provides, conveniently letting us implement an LRU cache for basis-ID pairs. We can also send updates regarding ID-basis pairs to other ZipLine instances out-of-band. The drawback of this approach is that updates take a longer time to effectively apply (*cf.*, section 8.7.2).

8.7. Evaluation

Our setup leverages an Edgecore Wedge100BF-32X programmable switch [49]. It is connected at 100 Gbit/s to two Dell PowerEdge R7515 servers through NVIDIA ConnectX-5 network cards interfaced with PCI Express 3.0 x16. Each server has an AMD EPYC 7302P processor, 32 GiB of RAM and runs Ubuntu 20.04.1. We tune each server with the `mlnx_tune` utility in *high throughput* mode. Unless stated otherwise, each measurement is repeated 10 times, and we show the average and the 95 % confidence interval.

8.7.1. Choice of parameters

It is possible to set several parameters in ZipLine. Three of them pertain to Hamming codes: k, n and m (as introduced in section 8.2.3). Specifically, the values of k and n are strongly linked to the value of m by formulas, so only m can in fact be freely selected. Since the Tofino platform has explicit byte-alignment constraints on header fields, every value of m that is not a multiple of 8 requires us to include wasteful padding bits in packets. We select $m = 8$ which is the largest m that is a multiple of 8 and that fits hardware constraints.

Additionally, it is possible to independently choose the size of the short identifiers that replace the bases. This parameter dictates how many bases have to be cached by ZipLine. Again, byte-alignment constraints have to be adhered to when choosing this value, such that padding bits can be omitted. We require one additional bit to store the MSB of the raw data packet—its size, k, is always one below a multiple of 8. Therefore, this parameter also needs to be one below a multiple of 8 to omit padding bits. Akin to our choice of $m = 8$, we settle for the largest value that is one below a multiple of 8 and fits hardware constraints (especially in terms of resource usage in this case): 15 bit, allowing for $2^{15} = 32\,768$ cached bases.

8.7.2. Dynamic learning

We want to measure the consequence of our decision to move the code responsible for managing bases-ID pairs to the control plane (*cf.*, section 8.6). In the data plane, every stateful operation appears to be instantaneous and already applies to the next packet in the pipeline. The main drawback of involving the control plane relates to performance, as it is located away from the path that packets take.

In this experiment, we measure the time between the arrival of an unknown basis in the switch and the moment after which the basis is registered in the compression table, and compressed packets start to be produced. To do so, we repeatedly send the same data packet as fast of possible from one server to another. We capture packets on the destination server and measure the amount of time it takes between the arrival of the first packet of type (ii) and the arrival of the first packet of type (iii) (*cf.*, section 8.5 for the definitions of types). Our results indicate that it takes (1.77 ± 0.08) ms for ZipLine to record and apply a new basis-ID pair. During that time, packets that share the same basis stay uncompressed. This loss in compression efficiency is measured next.

8.7.3. Compression

The goal of this experiment is to assess the compression ratio that can be obtained by using ZipLine. We use two datasets, a synthetic and a real-world one. We engineered the synthetic dataset to be behaviorally close to typical readouts from a sensor. We generate 3 124 000 chunks of 256 bit (matching the parameters we chose), which are then converted to a PCAP trace [74] of Ethernet packets containing the chunks as payload.

The real-world dataset consists in a day of domain name system (DNS) queries at a 4000 users university campus [159]. To obtain the trace that we replay, we apply filters to the downloaded files to only keep queries of 34 B going to the main DNS resolver of the campus, excluding the DNS transaction identifier which is a random number.

We replay these traces to our switch and monitor which action ZipLine undertakes with the payload of each packet. We then deduce the payload size, as each action produces a packet type of a fixed size. The sum of all original chunks represents the baseline. In figure 8.3, the bars represent the total size of the payloads of all packets after they are transformed by ZipLine. We also show the numerical ratio to the baseline next to each bar. We measure four cases:

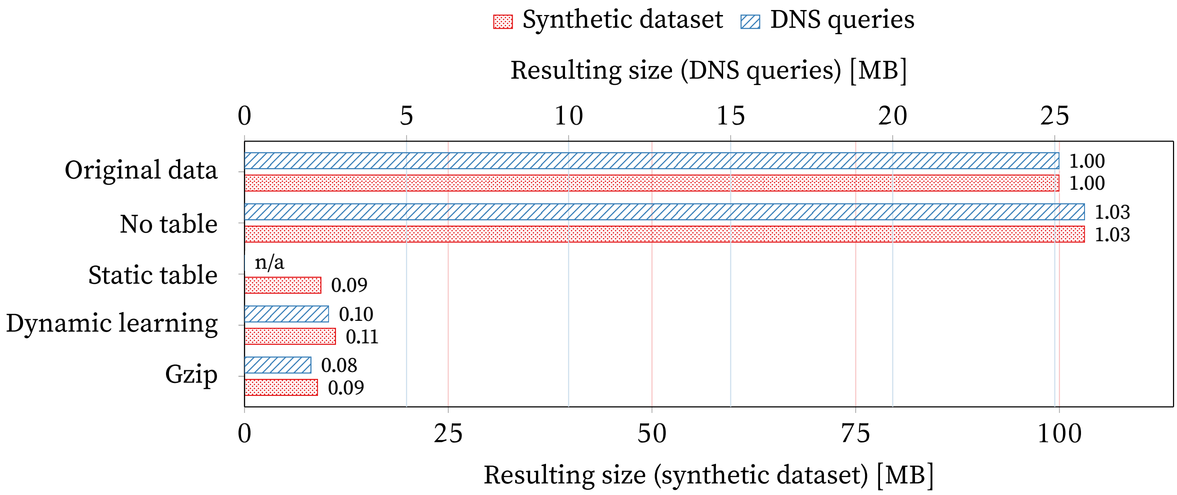

Figure 8.3.: Resulting payload size after traffic is processed with Gzip and ZipLine, without, with static-, and dynamically learned- compression table mappings.

No table The compression table stays empty.

Static table We pre-compute the basis of each payload and add a corresponding mapping in the compression table before we start the experiment.

Dynamic learning We start the experiment with an empty compression table, which is then automatically filled by ZipLine when unknown bases are encountered (real-world scenario).

Gzip We extract all payloads in a regular file that we compress with the `gzip` compression tool.

In theory, *no table* should be equal to the baseline as applying GD does not introduce additional bits. The overhead of 3 % is due to padding bits which are necessary to guarantee container alignment on the Tofino platform. We reckon that 8 such padding bits could be eliminated by an expert $P4_{16}$/TNA programmer. *Static table* represents the idealistic scenario where the basis of every packet payload is known beforehand. *Dynamic learning* represents a real scenario where the traffic is unknown to the switch prior to its arrival.

We see that ZipLine correctly learns and stores bases in its compression table, providing savings of 89 % in terms of bytes transmitted in the synthetic scenario, and up to 90 % in the DNS dataset. The compression ratio of ZipLine compares well with the `gzip` off-the-shelf compression utility (*circa* 20 % difference). It doubtlessly cannot be implemented on our hardware P4 target as it uses an algorithm (DEFLATE) that has an unbounded execution time.

The delta between the *static table* and *dynamic learning* cases in the synthetic scenario is first due to the fact that one packet with a payload mapping to each basis must be transmitted without compression to let the recipient know about it. Second, a couple milliseconds are needed between the time a packet with an unknown basis arrives in the switch, and the mapping to

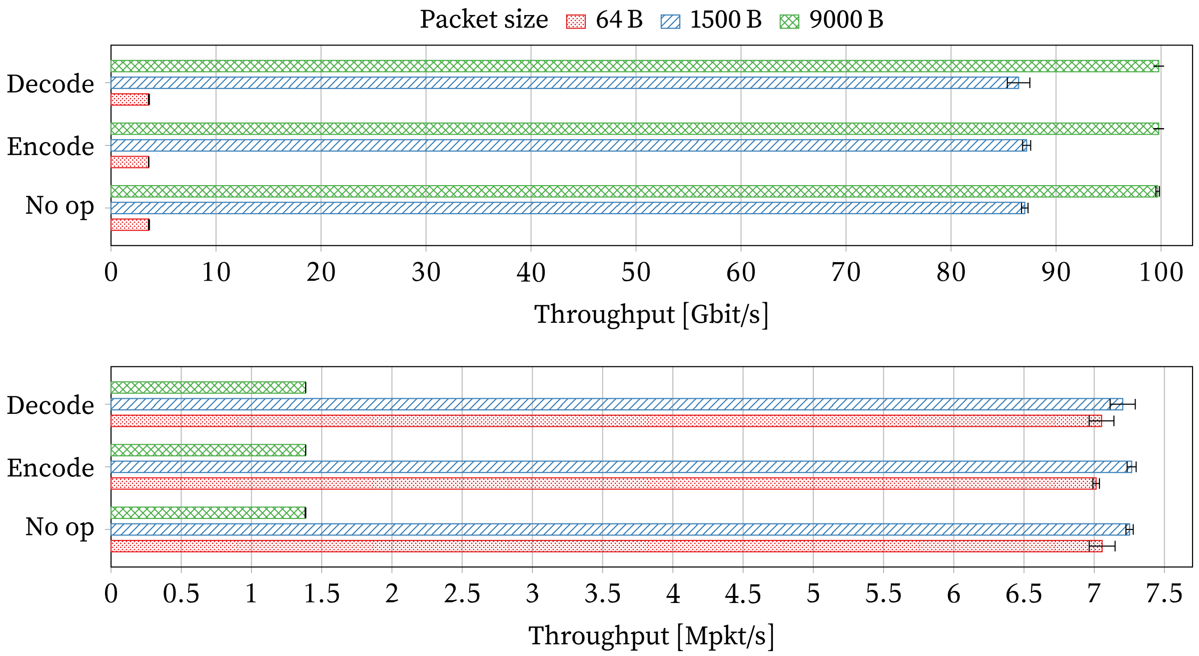

Figure 8.4.: Observed network throughput with the switch performing various operations on Ethernet frames of different sizes.

its basis becomes effective for subsequent packets (*cf.*, section 8.7.2), meaning that additional packets sharing the same basis stay uncompressed during this processing delay.

8.7.4. Raw performance

We now measure the raw performance of ZIPLINE using `raw_ethernet_` performance utilities [138] provided by NVIDIA Networking. We start by measuring the raw Ethernet throughput between 2 machines through the programmable switch. We transfer Ethernet frames of 3 common sizes for 10 s: the minimum frame size of 64 B, the standard 1500 B, as well as *jumbo* frames of 9000 B. The first scenario (*no op*) acts as the baseline, with the switch acting as a regular Ethernet switch. We then repeat the same measurements with the switch performing either the encoding or the decoding phase of ZIPLINE. Figure 8.4 shows our results. We notice that the claims put forwards by the vendor of our programmable switch are kept; namely that any P4$_{16}$ program that successfully compiles for the Tofino platform performs at line speed, as long as it does not make use of recirculation, packet duplication, *etc*. The figures for 64 B and 1500 B packets are bottlenecked at around 7 Mpkt/s by the server generating the traffic. In theory, the full line rate of 100 Gbit/s can also be reached with these smaller packets, as per the 4.7 Gpkt/s figure quoted in the datasheet of the switch [49].

Subsequently, we evaluate the latency by having one server send packets to itself via the programmable switch. As for the previous experiment, the switch performs regular Ethernet switching (*no op*), ZIPLINE encoding, or ZIPLINE decoding. We then measure the round-trip

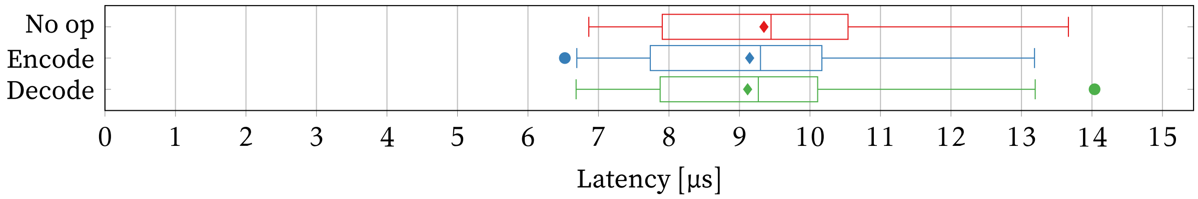

Figure 8.5.: Observed end-to-end latency with the programmable switch performing various operations.

time (RTT). Results are shown in figure 8.5 and point towards the same findings than our throughput measurements: the addition of ZipLine to the $P4_{16}$ program has no noticeable effect on raw performance.

8.8. Summary

This chapter introduced ZipLine, an approach for in-network data compression that can operate at line rate with minimal delay. We adapt the concept of generalized deduplication (GD) to provide an efficient implementation and a mapping of the data that can significantly boost the benefits of limited dictionaries implemented in the switches. We rely on transformations based on Hamming codes, which can be implemented efficiently on the programmable switch through its support of CRCs. Our implementation shows that (de)compression at line rate (100 Gbit/s per port on our switch) is possible and that dynamic learning can be implemented without compromising throughput or compression gains. Replaying a real-world data trace showed that ZipLine is capable of reducing data usage by up to 90 %.

The use of the CRC module in Tofino switches opens the door to computation of more complex transformations, *e.g.*, BCH codes, by using different generator polynomial parameters [26]. With those, more chunks to be mapped to each basis, albeit at the cost of a larger deviation in bits.

Chapter 9.

Transparent disaggregation of non-volatile main memory

9.1. Introduction

Non-volatile main memory (NVMM) represents a radical change in the computer memory hierarchy, associating persistence, direct byte-addressability and excellent performance. The technology is available on recent server-grade Intel processors and known as Intel Optane Persistent Memory (PMem) [92]. The Cascade Lake family of processors supports PMem series 100, while newer Ice Lake CPUs support PMem series 200.[1]

Given its versatile capabilities, NVMM can be used (i) as a faster alternative to a solid state drive (SSD); (ii) as a denser alternative to random access memory (RAM), or (iii) as a completely new memory paradigm. This last proposition could represent a complete change in how computers operate. With current PMem modules offering 512 GiB of directly-addressable persistent memory, one could envisage to primarily use PMem as system memory, with RAM only serving a subsidiary role.

Another contemporary paradigm in data centers is *hyperconvergence*. In a nutshell, it consists in virtualizing as much hardware as possible, such that one can compose virtual machines (VMs) with granular amounts of each type of resource. Every type of resource is connected to the *converged* network, which also serves as access network. Remote direct memory access (RDMA) is an essential enabling piece of technology behind hyperconverged storage. It reduces overheads in terms of throughput and latency to a minimum by delegating memory accesses to the network cards themselves. RDMA over Converged Ethernet (RoCE)[2] and iWARP are network protocols that implement RDMA over Internet protocol (IP).

In this context, we envisage that hyperconvergence should also pertain to NVMM. Further, NVMM is the only persistent storage medium that is directly accessible through RDMA (without involving the CPU).

Our proposed solution, **NVSplit**, can transparently disaggregate remote NVMM accesses across several servers through a Tofino programmable switch. NVSplit can split and/or mirror RDMA read and write requests across several servers containing PMem, respectively improving performance and reliability. Naturally, the use of RDMA to access the NVMM pool

[1] Those processors also support SGX 2 but it is not possible to enable both features simultaneously.

[2] Pronounced "rocky".

allows any RDMA-compatible client to claim the benefits of NVMM. We implement our prototype in a way that it transparent to client and servers thanks to the flexibility of the P4 language.

9.2. Related work

Several research efforts have attempted to use NVMM over the network.

Kalia *et al.* [99] show that the byte-addressability of NVMM and its performance make it a candidate of choice for RDMA. They highlight the pitfalls that one should be aware of when accessing PMem over the network.

Simurgh [133] and Octopus$^{(+)}$ [124, 188] are NVMM-enabled distributed filesystems. Octopus^{+} additionally uses RDMA for data transfers between servers. Both the clients and servers run specialized software in order to keep track of metadata to ultimately locate actual data. On the contrary, thanks to in-network processing, NVSPLIT does not need special support from the client, and can operate on raw NVMM (with or without a filesystem on top). Further, we can avoid all performance bottlenecks as the programmable switch always processes packets at line-speed, unlike a regular server.

In a different context, Kim *et al.* [105] initiate RDMA connections directly from the data plane of a programmable switch in order to augment the capacity of its stateful memory. Our work also initiates RDMA connections from the switch, but (i) from the control plane, and (ii) only as a preparatory step to later transform packets from external RDMA connections.

9.3. Non-volatile main memory persistence model

A crucial aspect to take into account when accessing NVMM over RDMA is the persistence model. When one uses RDMA to access RAM, it is important to know when a transfer is successful, *i.e.*, that the other party has successfully received the data. With NVMM, however, guaranteeing the successful transmission of data is not enough. The important element to consider is whether the data has been successfully *persisted* in the NVMM itself, *i.e.*, that the data will survive a sudden power loss. One-sided RDMA write operations only guarantee the former. A clever workaround is the appliance persistency method (APM) [47] which consists in performing an 8 B *read* operation after a *write* that we need to confirm. The ordered nature of RDMA work request queues [81] guarantees that the read operation will only return after the written data has been persisted in NVMM.

Another important point to take into account is Data Direct I/O (DDIO). It allows the network card to write into CPU caches to improve performance in some cases, as data is then ready to be processed by an application. In our use-case, however, the servers are only used as NVMM storage nodes; they never do any data processing. We can disable DDIO to ensure that the network card directly writes to NVMM, guaranteeing the aforementioned persistence model while cutting any detour short.

9.4. Concept

The idea behind NVSPLIT is to use an array of servers with PMem as one pool of NVMM shared across multiple clients in the datacenter. Existing NVMM over RDMA-compatible equipment and applications need to work flawlessly with NVSPLIT to make sure that adoption is straightforward. Aside from abstracting RDMA connections to NVMM servers, we want to provide additional advantages, such as better reliability and performance. Of note is that the throughput of PMem is inferior to that of 100 or 400 Gbit/s Ethernet connections [97].

As such, we specify four modes of operation:

1-to-1 Decoupling of clients and servers.

1-to-*n* mirroring Transparent data replication across multiple NVMM servers to increase reliability (similar in spirit to level 1 of a redundant array of independent disks (RAID)).

1-to-*n* splitting Round-robin data striping to different servers to provide better performance at the expense of reliability (similar in spirit to RAID 0).

1-to-*n* splitting+mirroring Round-robin striping overlaid on data replication to provide better performance and reliability at the same time (similar in spirit to RAID 10).

Each server will expose its NVMM range to the network. The clients and servers are all connected to a programmable switch that will transparently intercept RDMA requests and replies. It will transform these accordingly to provide the illusion that each machine only discusses with one other machine. In 1-to-*n* splitting mode, the switch will alter memory addresses to ensure that the servers' memory ranges are used contiguously.

We identify the following challenges to build such a system:

C1 Gain a deep understanding of the network protocols involved.

C2 Validate the feasibility of the project.

C3 Determine how to transparently intercept RDMA connections.

C4 Develop a suitable program that fits the constraints of our programmable switch.

C5 Test and refine NVSPLIT using existing RDMA-compatible applications.

9.4.1. RDMA decoupling

The main problem that we need to solve is how to decouple the client and server of an RDMA connection. Normally, RDMA always involves one client performing operations against one server. With NVSPLIT, the programmable switch will act as a *virtual* server towards the client and as a *virtual* client towards the servers. More specifically, it will transform, duplicate and redirect individual packets coming from each stakeholder, in such a way that all of them think that they are interacting with the switch itself.

The RoCE v2 protocol C1

We start by understanding our chosen RDMA protocol, RoCE v2, by reading through its comprehensive specification [81]. The protocol allows client machines to access remote NVMM with minimal overhead, and is supported by network cards from several vendors [31, 89, 139]. As we are going to intercept and modify network packets at a low level, we are interested in the format of RoCE packet themselves, and not necessarily in the application programming interfaces (APIs) that a program uses to control the behavior of the network card.

In essence, RoCE v2 is an encapsulation of InfiniBand in user datagram protocol (UDP) packets. Packet routing and switching uses standard Ethernet and IP mechanisms [80, 146, 42]. Above, InfiniBand is responsible for packet ordering, basic security mechanisms, and RDMA semantics.

Packet ordering is achieved with a per-connection counter: the packet sequence number (PSN). Each new request from the client increments the PSN, and each reply shares the PSN with its associated request. When a reply involves multiple packets (*e.g.*, RDMA read reply), the first reply packet shares the PSN of the request, and each subsequent reply packet increments the PSN. The following request will use the PSN that is immediately superior to the last reply packet. Acknowledgments can be in standalone packets or piggybacked, and tell the other end that all packets with a smaller or equal PSN have been received and processed. In case of re-transmissions, the PSN of the original packet is reused.

Each RDMA request needs to indicate an `R_Key`, which is a basic security mechanism that ensures that the target memory can only be accessed by entities that are authorized. The `R_Key` is a 32 bit key shared between the server and the client and transmitted in clear-text with each RDMA request. In the context of NVSplit, we can fairly easily circumvent this security mechanism by mounting the equivalent of a man-in-the-middle (MitM) attack.

InfiniBand defines several *operations*. Taking only one-sided RDMA operations into account, there exist two: `RDMA READ` and `RDMA WRITE`. Read operations always use *reliable* connections or datagrams, while write operations can use an *unreliable* connection.

Both read and write operations carry data in the same way. First, it is split in as many packets as necessary, making chunks that are of the largest power of 2 that fits the MTU, up to 4096 B (*e.g.*, 1024 B per packet for an MTU of 1500 B). If a single packet can carry the whole payload, a packet of type `Only` is used. Otherwise, the first packet is of type `First` and the last of type `Last`, with as many `Middle` packets in between as required.

A write operation is inherently uni-directional: the client can write at any time. The `First` or `Only` packet contains the virtual address (VA) of the first byte in memory, the `R_Key`, and the total data length of the operation. `Middle` and `Last` packets only contain data, plus general InfiniBand metadata. On the other hand, reading is a two-step operation. First, the client requests data from the server using a `READ Request` packet; it contains the same information as `WRITE First` and `WRITE Only` packets: VA, `R_Key`, and total data length. The server replies with `READ Response` `Only`, `First`, `Middle`, and `Last` packets as explained above.

Bootstrapping the values that the client needs to indicate in its requests happens during the initialization of the connection. On the wire, three messages are exchanged. First, the client sends a *Request for Communication* packet to the server, indicating its own information. The server replies with a comparable packet containing its own information (or outright rejects the connection request). Finally, the client acknowledges that the connection is ready with a *Ready To Use* packet. All three packets contain a field for private data that allows the application above to piggyback its own initialization information. Notwithstanding this standard connection establishment protocol, we observed that various RoCE-based applications use custom-made initialization protocols based on InfiniBand `SEND` operations, or even separate transmission control protocol (TCP) connections.

Feasibility C2

So far, we notice that RoCE consists of headers that are typical of a well-defined network protocol. Also, payload lengths exceeding one packet are always aligned to the same power of two for a particular connection. Those characteristics indicate that we should be able to parse and transform RoCE packets in P4 on TNA.

However, there is one field that is located past the payload: the invariant CRC (ICRC). It provides a checksum for a series of InfiniBand fields. When conforming to the protocol, this field stays *invariant* between endpoints. However, as we are going to modify some of the fields that the checksum protects, we need to change it accordingly for the endpoints to accept our modified packets. Unfortunately, it is not feasible to access this field using the programmable switch due to its location past the payload. Fortunately, there exists a way to disable ICRC validation on some RDMA network cards [136]. The loss in potential reliability is counterbalanced by a similar CRC specified in the Ethernet protocol [80], which RoCE depends on.

Transforming RDMA packets in the network C3

In figure 9.1, we see an example of the transformations required to split a write operation to two servers. We show how a three-packet long write operation to the *virtual* server materialized by the programmable switch is transformed to two distinct write operations to two servers.

Some transformations, like the `R_key`, consist in replacing a value with another. Once the connection is established, the value stays constant throughout the connection. In other cases, the value to insert in the transformed packet needs to be learned from the `First` packet. In this latter case, we will have to use stateful memory to remember the value from one packet to the next.

Further, most values require non-trivial computations (considering they need to fit the constraints of a programmable switch) to obtain the right value that makes the packet indistinguishable from a 1-to-1 RDMA operation. *E.g.*, computing the right PSN happens on a per-connection basis due to the different number of packets that each machine can exchange with the switch. At the same time, the computation also needs to take retransmissions into account, as well as assign the right PSN in the other direction for acknowledgments and read replies.

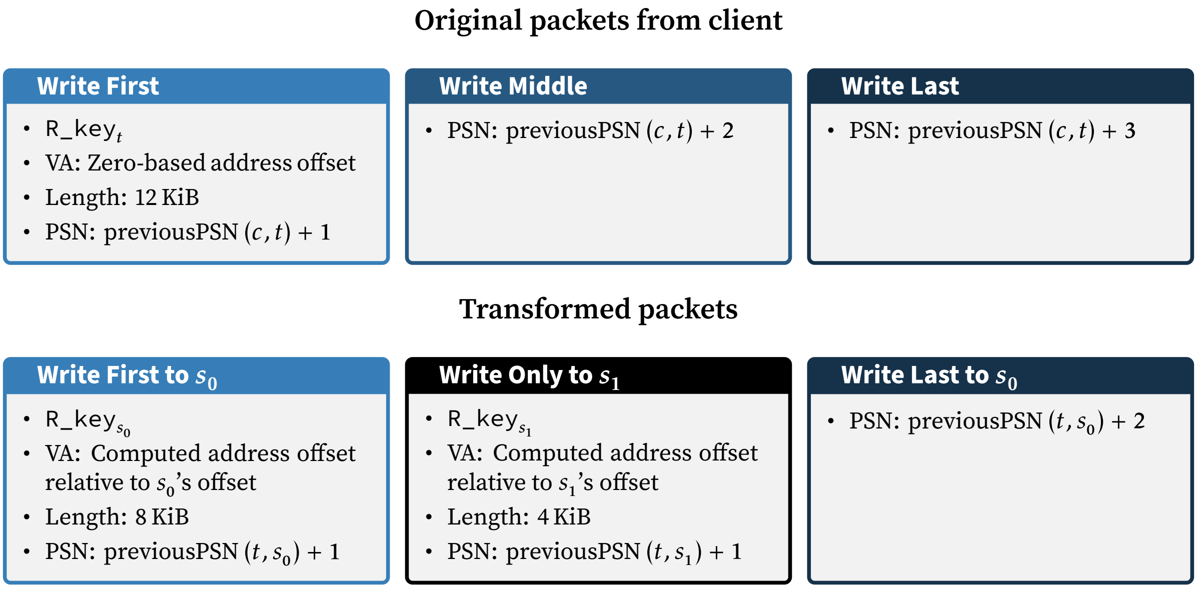
Original packets from client
Write First
R_key$_t$
VA: Zero-based address offset
Length: 12 KiB
PSN: previousPSN $(c, t) + 1$
Write Middle
PSN: previousPSN $(c, t) + 2$
Write Last
PSN: previousPSN $(c, t) + 3$
Transformed packets
Write First to s_0
R_key$_{s_0}$
VA: Computed address offset relative to s_0's offset
Length: 8 KiB
PSN: previousPSN $(t, s_0) + 1$
Write Only to s_1
R_key$_{s_1}$
VA: Computed address offset relative to s_1's offset
Length: 4 KiB
PSN: previousPSN $(t, s_1) + 1$
Write Last to s_0
PSN: previousPSN $(t, s_0) + 2$

Figure 9.1.: Transformation of RDMA packets for a 12 KiB 1-to-2 split write operation. c: client, $\{s_0, s_1\}$: servers, t: programmable switch.

9.5. Implementation

Our implementation revolves around the Intel Tofino programmable switch (*cf.*, section 3.2) for its versatility and uncompromising performance. The pool of NVMM servers can use either Intel Optane PMem series 100 or PMem series 200, but all servers should use the same generation. Due to the workaround discussed in section 9.4.1, only NVIDIA ConnectX network cards [139] can effectively be used with NVSplit.

9.5.1. Data plane C4

Implementing NVSplit in the data plane is challenging due to the pipelined nature of the programmable switch. We need to devise algorithms that can compute the various values that we need to transform. In particular, we need to take the constraints of P4 into account, as well as the hardware constraints of our Tofino programmable switch. For instance, we use *registers* (*cf.*, section 3.2) to have access to information that was present in an earlier packet. However, storage and retrieval operations of a particular register *must* happen in the exact same *stage* of the pipeline. Depending on the order of various operations, an impossible dependency cycle can be created. At that point, one needs to rethink their P4 program in a different way.

Flexibility, seamlessness, and performance are cornerstones of the design of NVSplit. We want our data plane to reflect that, so we make sure that all features are compiled in a single P4 program, with each being activated through P4 tables. For similar reasons, we need to process RDMA packets on-the-fly within the data plane. We refrain from using recirculation to obtain the best possible performance.

Our P4 program works as such. First, we program both parsers (ingress and egress) to dissect packets down to RETH and AETH RoCE headers. In the ingress, packets that are not RoCE are switched regularly according to the Ethernet protocol. RoCE connection initialization packets are always forwarded to the control plane, as they contain all the information that the control needs to fill in the relevant tables. Conversely, packets that are injected by the control plane (whether RoCE or not) are switched regularly, as we expect the control plane to produce ready-to-send packets.

After those special predicaments are addressed, we are left with RoCE packets coming from physical ports. The first action is to match them against their senders to determine which mode of operation (*cf.*, section 9.4) is applicable.

1-to-1

In 1-to-1 mode, we only apply basic transformations such as replacing the R_Key, and add fixed offsets to the PSN and VA. For such transformations, we can use a P4 *table* and fill in the correct value using the control plane.

Determining where to send each packet only depends on its direction. All request packets go to the server and all replies to the client. Predictably, we need to transform the addresses and ports indicated in lower protocols, *i.e.*, IP and Ethernet.

1-to-n mirroring

Mirroring is a more involved operation, as some of the processing must happen in the egress (*cf.*, section 3.1), *e.g.*, offsetting the PSN on request packets.

We need to keep track of which requests have been acknowledged, to only send an acknowledgment to the client after all n servers have successfully performed the operation. We use registers to count how many acknowledgments a client-facing PSN has received. We release an acknowledgment packet to the client only when the counter is equal to the number of servers.

1-to-n splitting

Splitting is the most intricate mode to implement due to the added asynchronicity on the number of packets between participants of the split connection. While the client will see all packets, each server only sees a subset depending on n and the length of the RDMA operations. Computing PSNs becomes a stateful process instead of an addition.

Unlike the other modes, the destination of a client to server RDMA packet is determined by its *VA parity*. To make this possible, we limit n to powers of two. In the case of writing, we statefully compute the zero-based client-facing VA for each packet, and extract $\log_2 n$ bits from the address. We take those at an offset that depends on the MTU, *e.g.*, from the $\log_2 4096 = 12$th bit when the maximal data length is 4096 B (*cf.*, section 9.4.1). A table maps every identifier in $[0, n)$ to one destination server. To prevent gaps in the memory range of each server, we then shift the most-significant bits of the VA indicated in the original request to the right to fill the bits that we extracted to determine the destination server.

`WRITE` packets that are or become `First` packets on the server-side need to have their data length field adjusted accordingly. *E.g.*, with 2 servers, the original length can be halved, halved plus one full packet (MTU-dependent), or halved plus one partial packet, depending on n, the original data length and the computed VA parity.

`READ` requests are processed similarly, albeit they first need to be duplicated to the right number of servers depending on the indicated data length. Each replica of the reading request is then processed in the egress to customize most fields, again using the VA parity, dividing the data length among servers. Each server then sends back the data requested at its own pace, and NVSPLIT will compute the right PSN for each reply so that the client can reassemble the data in the right order.

As exemplified in section 9.4.1, packet types `FIRST`, `MIDDLE`, `LAST`, and `ONLY` often need to be changed to another type. Once again, we use registers to compute whether a packet is the first and only one of the transaction (`ONLY`), the first with more to follow (`FIRST`), the last of the transaction (`LAST`), or any other packet (`MIDDLE`). Of note is that only `FIRST` and `ONLY` packets contain an `RETH` header [81]. Transforming a `MIDDLE` packet to one of those requires to create such a header from scratch, using tables and registers accordingly.

9.5.2. Control plane

The control plane acts as a companion to the data plane. It fills in table entries and initializes register values as required. We implement the control plane in Python, and connect it to the data plane through Google remote procedure calls (gRPC) using BfRt.

Clients in effect initialize their RDMA connections to the control plane itself. In turn, the control plane acts as an RDMA client towards n servers, according to the mode of operation in effect. We use Scapy [25] as a framework to decode and craft RoCE initialization packets. When the control plane receives a *Request for Communication* packet (*cf.*, section 9.4.1) from a client, it completes the initialization by blending in relevant values learned from the initialization replies coming from the servers. At the same time, the control plane fills in the P4 tables and registers to allow the data plane to seamlessly transform RDMA read and write requests from one connection to the other. With the initialization phase over, the control plane is not involved anymore until one party disconnects. At that point, it propagates the disconnection request to all parties and clears all table entries that pertain to the connection.

9.5.3. Client and server C5

NVSPLIT servers each expose an array of NVMM through RDMA. They listen on connections coming from the control plane, and handle it like any other reliable RDMA connection. Clients similarly connect to the control plane in a standard way. In that sense, NVSPLIT is transparent to all parties. The only condition to use a particular client-server pair is to adapt the control plane to correctly extract the relevant information from the initialization packets.

Thanks to the simplicity of Python and Scapy, this is easily done. Existing software based on librpma [16], *e.g.*, fio (*cf.*, section 9.6.2), is already supported by the current version of NVSPLIT.

To further simplify the use of NVSPLIT, we provide a simple server to use as a basis for any custom client. Simultaneously, we develop a client library that replaces key Linux file operations (*i.e.*, `open`, `close`, `read`, and `write`) to operate on remote NVMM servers through RDMA. Most legacy applications can hence work with NVSPLIT simply by preloading our library [117].

9.6. Results

9.6.1. Experimental setup

We deploy NVSPLIT on an Edgecore Wedge 100BF-32X programmable switch. We use 2 Supermicro X11 DP servers, each with 2 Intel Xeon Gold 5215 CPUs, 128 GiB of RAM, and 2 modules of 128 GiB of Intel Optane PMem series 100. Our clients consist of Dell PowerEdge R7515 machines with an AMD EPYC 7302P CPU and 32 GiB of RAM.

All machines are equipped with NVIDIA ConnectX-5 network cards interfaced through PCI Express 3.0 x16 links. Each card is directly connected to the programmable switch using 100 Gbit/s Ethernet.

9.6.2. Micro-benchmark

We assemble a micro-benchmark using the fio benchmarking tool [13] and its librpma-based storage engine [16]. With this experiment, we want to verify that we obtain performance metrics that match our expectations in terms of throughput. The benchmark only performs write operations, with the persistence model guaranteed by APM.

We first run the experiment with one client writing to a single server without NVSPLIT. Then, we set the programmable switch in 1-to-2 split mode, and run the benchmark again. We report a write throughput of 3374 MB/s for the baseline, while two-server data-splitting allows the throughput to rise to 6207 MB/s, a speedup of 1.84.

With this small-scale measurement, we confirm that the idea behind NVSPLIT is practical in the real world. We show that existing applications can be used unmodified with NVSPLIT, and that it is possible to obtain faster-than-native PMem performance through split RDMA writes.

9.7. Summary

In this chapter, we presented NVSPLIT, an in-network NVMM disaggregator. By carefully studying the specification governing our chosen RDMA protocol, RoCE, we manage to intercept and modify RDMA packets in a way that is transparent to the client and servers. We use this capability to split and mirror RDMA connections similarly to RAID levels 0 and 1. We implement NVSPLIT on the first-generation Intel Tofino programmable switch without using any kind of recirculation. Therefore, the performance of NVSPLIT is only limited by clients and/or servers. We assemble a micro-benchmark to show that we can indeed achieve faster-than-native performance with PMem series 100 thanks to data striping across two servers.

Despite the immense potential behind the concept of NVMM, Intel has recently announced that it will stop to make Optane PMem products [147]. We believe that the technology still has lots of untapped potential, as we have shown with NVSPLIT. However, it is possible that the end of the PMem product will sooner or later lead to the end of practical (*i.e.*, not simulated) systems research on NVMM, unless an equivalent product comes on the market in the future.

9.7.1. Future work

Implementing NVSPLIT proved to require considerable efforts, but is not yet complete. Combining splitting and mirroring *à la* RAID-10 is left for future work. Similarly, provisions exist for 1-to-*n* splitting to more than 2 servers, but the complete implementation is not yet finished.

With all features implemented, we will perform a full-scale evaluation of NVSPLIT, hopefully confirming the performance improvement measured by our micro-benchmark.

Chapter 10.

Conclusion

Summary

Throughout our thesis, we investigated how recently-introduced hardware-assisted security and networking technologies can be used to solve contemporary issues related to cloud-based systems.

On paper, trusted execution environments (TEEs) represent a substantial innovation. In particular, the attestation mechanism provided by Intel Software Guard Extensions (SGX) completely changes the trust model to apply when offloading computing tasks to a third party. It becomes possible to distrust the third party as programs protected by a TEE are presumably shielded from powerful attackers, including those with physical access to the machine that the program runs on.

In chapter 4, we started by implementing STRESS-SGX, an SGX-specific *stressing* tool that also serves as a benchmarking tool. We used it and other tools to measure the performance of Intel SGX and AMD Secure Encrypted Virtualization (SEV). We saw that TEEs can indeed run programs without overheads in some cases. Offering lesser security guarantees, AMD SEV in particular imposes very little performance penalty to the programs it protects. On the other hand, Intel SGX better protects against sophisticated attacks. Consequently, the programs that it shields are more prone to performance overheads. Several security flaws against Intel SGX have been found by other researchers during the course of this thesis, but the majority of them have been fixed or mitigated to our knowledge. However, those fixes and mitigations have usually led to severe performance penalties: we measured slowdowns of up to 3.8× in our benchmarks. Nevertheless, with careful use, it remains possible to protect applications with SGX with virtually no overheads, unlike similar software-only shielding techniques, *e.g.*, homomorphic encryption.

A sizable disadvantage of SGX 1 is the very limited size of its reserved memory, the enclave page cache (EPC). Consequently, we designed and implemented a container orchestrator that can deploy SGX-enabled containers across a heterogeneous cluster (*cf.*, chapter 5). The scheduler takes the requests of SGX and non-SGX containers concurrently to efficiently use the limited EPC of each machine. It is crucial to stay within the limits of the EPC to prevent considerable performance overheads, which could be especially problematic in a shared infrastructure.

With that knowledge and our new tools at hand, we designed A-SKY, a privacy-preserving group data sharing system (*cf.*, chapter 6). Thanks to the security guarantees provided by SGX, we found ways to modify an existing cryptographic scheme to reduce its complexity by

3 orders of magnitude. Our implementation only requires SGX on a few server machines. By using the aforementioned SGX-aware orchestrator, we are able to package and deploy the various components of A-SKY as micro-services. We can effectively scale up or down the system in a practical way as our evaluation showed that most operations scale linearly with the number of servers.

Apart from security and privacy issues, the growing amount of network traffic represents another contemporary issue. With ENDBOX (*cf.*, chapter 7), we proposed a potential way to use underused client resources to perform network security functions within SGX enclaves. As packet analysis happens on the clients themselves, it becomes possible to analyze encrypted traffic without using classical techniques that amount to a man-in-the-middle attack. We showed that our proposal is practical, in particular in an enterprise setting.

With ENDBOX, we designed a complete system to alleviate companies from needing expensive centralized networking appliances by pushing workloads to clients. A recent technique is instead to perform computations within the network itself. Data plane-programmable switches offer the opportunity to apply virtually any treatment to packets within the network, at no extra cost.

In chapter 8, we explained that we can adapt the generalized deduplication (GD) technique to fit the resources of an Intel Tofino programmable switch. In line with the promises of programmable switches, our system, ZIPLINE, can perform compression and decompression at line-rate without any extra latency. We found out that it performs exceptionally well in a realistic scenario, as we could compress a stream of domain name system (DNS) packets by 90 % in real-time.

Finally, in chapter 9, we used the programmable switch to disaggregate remote accesses to a new type of hardware: non-volatile main memory (NVMM). Similarly to what we observed with ZIPLINE, the mere fact that we managed to make our program, NVSPLIT, fit the resources of the programmable switch imply that packet processing happens at line-rate. As NVSPLIT can *split* a connection to multiple servers, and today's network speeds are faster than that of Intel Optane Persistent Memory (PMem), we can achieve faster-than-native NVMM accesses. Furthermore, NVSPLIT intercepts standard RDMA over Converged Ethernet (RoCE) connections and transforms them such that it stays completely transparent to all participants.

Perspectives and future work

When possible, we freely released the software associated to our contributions. Other people have already used parts of our work to further their own.

In each chapter, we showed that one can use new hardware-assisted techniques in an inventive way to solve contemporary problems more efficiently or securely than with pure software-based solutions. However, in contrast with software-based solutions, special knowledge is required to use hardware-assisted techniques. To begin with, it is important to know the limitations of each technique to make the best use of it. Second, using special tools, software development kits (SDKs) and programming languages is often required to run one's program on a hardware contraption. Nevertheless, some solutions are only possible by using those new techniques. For instance, changing core network protocols routinely takes years or even decades. Should programmable switches become commonplace, such improvements could smoothly happen through *software* upgrades instead of *hardware* replacements.

Regrettably, most hardware-assisted techniques are vendor-specific. While most vendors of programmable switches form a consortium around the P4 programming language, there is nothing similar for TEEs. Worse, when a single vendor offers a specific type of hardware, its cancellation—as is happening to NVMM with Intel stopping the production of Optane PMem products—can mean the end of a technique, without any suitable replacement.

On the other hand, new versions of a technique can be promising. For instance, SGX 2 increases the maximal size of the EPC from 128 MiB to 512 GiB. Future work could show how the various SGX-enabled systems presented in this thesis behave under this revision of SGX.

Chapter 9 is incomplete as some variants of our system are not completely implemented. Future work will rectify that such that a thorough evaluation of the system can be completed.

Publications

2018

Sébastien Vaucher, Valerio Schiavoni, and Pascal Felber. 2019. Stress-SGX: load and stress your enclaves for fun and profit. In *Networked Systems - 6th International Conference, NETYS 2018, Essaouira, Morocco, May 9-11, 2018, Revised Selected Papers* (Lecture Notes in Computer Science). Andreas Podelski and François Taïani, (Eds.) Vol. 11028. Springer, 358–363. DOI: 10.1007/978-3-030-05529-5_24

David Goltzsche, Signe Rüsch, Manuel Nieke, Sébastien Vaucher, Nico Weichbrodt, Valerio Schiavoni, Pierre-Louis Aublin, Paolo Costa, Christof Fetzer, Pascal Felber, Peter R. Pietzuch, and Rüdiger Kapitza. 2018. EndBox: scalable middlebox functions using client-side trusted execution. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*. IEEE Computer Society, 386–397. DOI: 10.1109/DSN.2018.00048

Stefan Contiu, Rafael Pires, Sébastien Vaucher, Marcelo Pasin, Pascal Felber, and Laurent Réveillère. 2018. IBBE-SGX: cryptographic group access control using trusted execution environments. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*. IEEE Computer Society, 207–218. DOI: 10.1109/DSN.2018.00032

Sébastien Vaucher, Rafael Pires, Pascal Felber, Marcelo Pasin, Valerio Schiavoni, and Christof Fetzer. 2018. SGX-aware container orchestration for heterogeneous clusters. In *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*. IEEE Computer Society, 730–741. DOI: 10.1109/ICDCS.2018.00076

Christian Göttel, Rafael Pires, Isabelly Rocha, Sébastien Vaucher, Pascal Felber, Marcelo Pasin, and Valerio Schiavoni. 2018. Security, performance and energy trade-offs of hardware-assisted memory protection mechanisms. In *37th IEEE Symposium on Reliable Distributed Systems, SRDS 2018, Salvador, Brazil, October 2-5, 2018*. IEEE Computer Society, 133–142. DOI: 10.1109/SRDS.2018.00024

2019

Stefan Contiu, Sébastien Vaucher, Rafael Pires, Marcelo Pasin, Pascal Felber, and Laurent Réveillère. 2019. Anonymous and confidential file sharing over untrusted clouds. In *38th Symposium on Reliable Distributed Systems, SRDS 2019, Lyon, France, October 1-4, 2019*. IEEE, 21–31. ISBN: 978-1-7281-4222-7. DOI: 10.1109/SRDS47363.2019.00013

2020

Franz Gregor, Wojciech Ozga, Sébastien Vaucher, Rafael Pires, Do Le Quoc, Sergei Arnautov, André Martin, Valerio Schiavoni, Pascal Felber, and Christof Fetzer. 2020. Trust management as a service: enabling trusted execution in the face of byzantine stakeholders. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*. IEEE, 502–514. DOI: 10.1109/DSN48063.2020.00063

Sébastien Vaucher, Niloofar Yazdani, Pascal Felber, Daniel E. Lucani, and Valerio Schiavoni. 2020. ZipLine: in-network compression at line speed. In *CoNEXT '20: The 16th International Conference on emerging Networking EXperiments and Technologies, Barcelona, Spain, December, 2020*. Dongsu Han and Anja Feldmann, (Eds.) ACM, 399–405. DOI: 10.1145/3386367.3431302

Bibliography

[1] Advanced Micro Devices, Inc. AMD EPYC datacenter processor launches with record-setting performance, optimized platforms, and global server ecosystem support. AMD Press Releases, (20 June 2017). Retrieved 13 June 2022 from https://ir.amd.com/news-events/press-releases/detail/773/amd-epyc-datacenter-processor-launches-with.

[2] Advanced Micro Devices, Inc. 2020. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more. White paper. (Jan. 2020). Retrieved 20 June 2022 from https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[3] Alfred V. Aho and Margaret J. Corasick. 1975. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18, 6, 333–340. DOI: 10.1145/360825.360855.

[4] Tiago Alves and Don Felton. 2004. TrustZone: integrated hardware and software security. *Information Quarterly*, 3, 18–24, 4. Retrieved 21 June 2022 from https://web.archive.org/web/20220621084709/https://docplayer.net/storage/64/51242536/1655804612/wHpF4YF_cquwF32D6hfmQQ/51242536.pdf.

[5] Amazon Web Services, Inc. 2017. Alexa top sites. (2017). Retrieved 19 Aug. 2022 from https://web.archive.org/web/20170716192359/https://aws.amazon.com/alexa-top-sites/.

[6] Amazon Web Services, Inc. 2022. *Amazon Elastic Container Service Documentation*. Retrieved 3 Mar. 2022 from https://docs.aws.amazon.com/ecs/index.html.

[7] Amazon Web Services, Inc. 2022. *Amazon Simple Storage Service Documentation*. Retrieved 23 Aug. 2022 from https://docs.aws.amazon.com/s3/index.html.

[8] Ittai Anati, Shay Gueron, Simon Johnson and Vincent Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In 2nd International Workshop on Hardware and Architectural Support for Security and Privacy. Special session. Retrieved 12 Aug. 2022 from https://www.intel.com/content/dam/develop/external/us/en/documents/hasp-2013-innovative-technology-for-attestation-and-sealing.pdf.

[9] Sebastian Angel and Srinath T. V. Setty. 2016. Unobservable communication over fully untrusted infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 551–569. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/angel.

[10] Michael Angelo, Boris Balacheff, Josh Benaloh, David Challener, Dhruv Desai, Paul England, David Grawrock, Bob Meinschein, Manny Novoa, Graeme Proudler, Jim Ward and Monty Wiseman. 2002. *Trusted Computing Platform Alliance. Main Specification*. Version 1.1b. Trusted Computing Group. https://trustedcomputinggroup.org/wp-content/uploads/TCPA_Main_TCG_Architecture_v1_1b.pdf.

[11] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, André Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Peter R. Pietzuch and Christof Fetzer. 2016. SCONE: secure Linux containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 689–703. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.

[12] Pierre-Louis Aublin, Florian Kelbert, Dan O'Keffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers and Peter Pietzuch. 2017. TaLoS: Secure and Transparent TLS Termination inside SGX Enclaves. Tech. rep. 17/5. Imperial College London. DOI: 10.25561/94936.

[13] [SW] Jens Axboe, Flexible I/O Tester 2022. Retrieved 8 Sept. 2022 from URL: https://github.com/axboe/fio.

[14] Michael Backes, Christian Cachin and Alina Oprea. 2006. Secure key-updating for lazy revocation. In *Computer Security - ESORICS 2006, 11th European Symposium on Research in Computer Security, Hamburg, Germany, September 18-20, 2006, Proceedings* (Lecture Notes in Computer Science). Vol. 4189. Springer, 327–346. DOI: 10.1007/11863908_21.

[15] Fred Baker, David L. Black, Kathleen Nichols and Steven L. Blake. 1998. Definition of the differentiated services field (DS field) in the IPv4 and IPv6 headers. RFC 2474. (Dec. 1998). DOI: 10.17487/RFC2474.

[16] Piotr Balcer, ed. 2022. The librpma library. Intel Corporation, (2022). Retrieved 7 Sept. 2022 from https://pmem.io/rpma/.

[17] Hitesh Ballani, Paolo Costa, Christos Gkantsidis, Matthew P. Grosvenor, Thomas Karagiannis, Lazaros Koromilas and Greg O'Shea. 2015. Enabling end-host network functions. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*. ACM, 493–507. DOI: 10.1145/2785956.2787493.

[18] Elaine Barker. 2020. Recommendation for key management, part 1: general. National Institute of Standards and Technology, (2020). DOI: 10.6028/NIST.SP.800-57pt1r5.

[19] Adam Barth, Dan Boneh and Brent Waters. 2006. Privacy in encrypted content distribution using private broadcast encryption. In *Financial Cryptography and Data Security, 10th International Conference, FC 2006, Anguilla, British West Indies, February 27-March 2, 2006, Revised Selected Papers* (Lecture Notes in Computer Science). Vol. 4107. Springer, 52–64. DOI: 10.1007/11889663_4.

[20] Mihir Bellare, Anand Desai, David Pointcheval and Phillip Rogaway. 1998. Relations among notions of security for public-key encryption schemes. In *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings* (Lecture Notes in Computer Science). Vol. 1462. Springer, 26–45. DOI: 10.1007/BFb0055718.

[21] Alysson Neves Bessani, Miguel Correia, Bruno Quaresma, Fernando André and Paulo Sousa. 2013. Depsky: dependable and secure storage in a cloud-of-clouds. *ACM Trans. Storage*, 9, 4, 12. DOI: 10.1145/2535929.

[22] Alysson Neves Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Ferreira Neves, Miguel Correia, Marcelo Pasin and Paulo Veríssimo. 2014. SCFS: a shared cloud-backed file system. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. USENIX Association, 169–180. https://www.usenix.org/conference/atc14/technical-sessions/presentation/bessani.

[23] John Bethencourt, Amit Sahai and Brent Waters. 2007. Ciphertext-policy attribute-based encryption. In *2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA*. IEEE Computer Society, 321–334. DOI: 10.1109/SP.2007.11.

[24] [SW] Timo Bingmann, Parallel Memory Bandwidth Benchmark / Measurement version 0.6.2, 12 Dec. 2013. Retrieved 20 June 2022 from URL: http://panthema.net/2013/pmbw/.

[25] Philippe Biondi et al. 2022. Scapy. Packet crafting for Python2 and Python3. Retrieved 8 Sept. 2022 from https://scapy.net/.

[26] Richard E. Blahut. 2003. *Algebraic Codes for Data Transmission*. Cambridge University Press. DOI: 10.1017/CBO9780511800467.

[27] Dan Boneh, Craig Gentry and Brent Waters. 2005. Collusion resistant broadcast encryption with short ciphertexts and private keys. In *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings* (Lecture Notes in Computer Science). Vol. 3621. Springer, 258–275. DOI: 10.1007/11535218_16.

[28] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese and David Walker. 2014. P4: programming protocol-independent packet processors. *Comput. Commun. Rev.*, 44, 3, 87–95. DOI: 10.1145/2656877.2656890.

[29] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter R. Pietzuch and Rüdiger Kapitza. 2016. SecureKeeper: confidential ZooKeeper using Intel SGX. In *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016*. ACM, 14. DOI: 10.1145/2988336.

[30] Broadcom Inc. 2019. *NPL - Network Programming Language Specification*. Version 1.3. Retrieved 31 Aug. 2022 from https://raw.githubusercontent.com/nplang/NPL-Spec/master/NPL%20spec%20version%201.3.pdf.

[31] Broadcom Inc. 2022. RDMA over Converged Ethernet feature in Ethernet NIC controllers. Retrieved 5 Sept. 2022 from https://techdocs.broadcom.com/us/en/storage-and-ethernet-connectivity/ethernet-nic-controllers/bcm957xxx/1-0/introduction/features_27/rdma-over-converged-ethernet--roce.html.

[32] Brocade Communications Systems, Inc. 2017. *Brocade 6520 Switch*. Retrieved 13 Sept. 2022 from https://docs.broadcom.com/doc/12379876.

[33] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom and Raoul Strackx. 2018. Foreshadow: extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. USENIX Association, 991–1008. https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[34] Stephen Checkoway and Hovav Shacham. 2013. Iago attacks: why the system call API is a bad untrusted RPC interface. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*. ACM, 253–264. DOI: 10.1145/2451116.2451145.

[35] Xiaoqi Chen. 2020. Implementing AES encryption on programmable switches via scrambled lookup tables. In *Proceedings of the 2020 ACM SIGCOMM 2020 Workshop on Secure Programmable Network Infrastructure, SPIN@SIGCOMM 2020, Virtual Event, USA, August 14, 2020*. ACM, 8–14. DOI: 10.1145/3405669.3405819.

[36] Stefan Contiu, Rafael Pires, Sébastien Vaucher, Marcelo Pasin, Pascal Felber and Laurent Réveillère. 2018. IBBE-SGX: cryptographic group access control using trusted execution environments. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*. IEEE Computer Society, 207–218. DOI: 10.1109/DSN.2018.00032.

[37] Stefan Contiu, Sébastien Vaucher, Rafael Pires, Marcelo Pasin, Pascal Felber and Laurent Réveillère. 2019. Anonymous and confidential file sharing over untrusted clouds. In *38th Symposium on Reliable Distributed Systems, SRDS 2019, Lyon, France, October 1-4, 2019*. IEEE, 21–31. ISBN: 978-1-7281-4222-7. DOI: 10.1109/SRDS47363.2019.00013.

[38] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*. ACM, 143–154. DOI: 10.1145/1807128.1807152.

[39] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura and Ricardo Bianchini. 2017. Resource central: understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 153–167. DOI: 10.1145/3132747.3132772.

[40] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, 86. https://eprint.iacr.org/2016/086.

[41] Michael Coughlin, Eric Keller and Eric Wustrow. 2017. Trusted Click: overcoming security issues of NFV in the cloud. In *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization, SDN-NFVSec@CODASPY 2017, Scottsdale, Arizona, USA, March 24, 2017*. ACM, 31–36. DOI: 10.1145/3040992.3040994.

[42] Dr. Steve E. Deering and Bob Hinden. 2017. Internet protocol, version 6 (IPv6) specification. RFC 8200. (July 2017). DOI: 10.17487/RFC8200.

[43] DevOps.com and ClusterHQ. 2016. Container market adoption. Survey. (June 2016). Retrieved 3 Mar. 2022 from https://web.archive.org/web/20170913060955/https://clusterhq.com/assets/pdfs/state-of-container-usage-june-2016.pdf.

[44] Colin Dixon, Hardeep Uppal, Vjekoslav Brajkovic, Dane Brandon, Thomas E. Anderson and Arvind Krishnamurthy. 2011. ETTM: a scalable fault tolerant network manager. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*. USENIX Association. https://www.usenix.org/conference/nsdi11/ettm-scalable-fault-tolerant-network-manager.

[45] Docker, Inc. 2021. Swarm mode overview. (25 Mar. 2021). Retrieved 3 Mar. 2022 from https://docs.docker.com/engine/swarm/.

[46] Danny Dolev and Andrew Chi-Chih Yao. 1983. On the security of public key protocols. *IEEE Trans. Inf. Theory*, 29, 2, 198–207. DOI: 10.1109/TIT.1983.1056650.

[47] [SW] Lukasz Dorau, Oksana Sałyk, Xiao Yang, Tomasz Gromadzki and Kacper Stefanski, librpma flush-related implementations 2022. Retrieved 7 Sept. 2022 from URL: https://github.com/pmem/rpma/blob/a061dcbd489768656dee651e9ca5a8ce34b29db7/src/flush.c#L38.

[48] Huayi Duan, Cong Wang, Xingliang Yuan, Yajin Zhou, Qian Wang and Kui Ren. 2019. LightBox: full-stack protected stateful middlebox at lightning speed. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. ACM, 2351–2367. DOI: 10.1145/3319535.3339814.

[49] Edgecore Networks Corporation. 2019. *DCS800 Data Center Switch. Wedge100BF-32X*. Datasheet. Retrieved 17 Mar. 2022 from https://www.edge-core.com/_upload/images/2021-138-DCS800_Wedge100BF-32X-R09-20211209.pdf.

[50] Eurostat. ICT usage in enterprises. Use of cloud computing. Retrieved 9 Sept. 2022 from https://ec.europa.eu/eurostat/databrowser/view/isoc_cicce_use/default/table.

[51] Christos Faloutsos, M. Ranganathan and Yannis Manolopoulos. 1994. Fast subsequence matching in time-series databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994*. ACM Press, 419–429. DOI: 10.1145/191839.191925.

[52] Markus Feilner. 2006. *OpenVPN: Building and integrating virtual private networks*. Packt Publishing Ltd. ISBN: 978-1-904811-85-5.

[53] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis and Haining Wang. 2017. ContainerLeaks: emerging security threats of information leakages in container clouds. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*. IEEE Computer Society, 237–248. DOI: 10.1109/DSN.2017.49.

[54] Aman Garg and A. L. Narasimha Reddy. 2004. Mitigation of DoS attacks through QoS regulation. *Microprocess. Microsystems*, 28, 10, 521–530. DOI: 10.1016/j.micpro.2004.08.007.

[55] William C. Garrison III, Adam Shull, Steven A. Myers and Adam J. Lee. 2016. On the practicality of cryptographically enforcing dynamic access control policies in the cloud. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 819–838. DOI: 10.1109/SP.2016.54.

[56] Blaise Gassend, G. Edward Suh, Dwaine E. Clarke, Marten van Dijk and Srinivas Devadas. 2003. Caches and hash trees for efficient memory integrity verification. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA'03), Anaheim, California, USA, February 8-12, 2003*. IEEE Computer Society, 295–306. DOI: 10.1109/HPCA.2003.1183547.

[57] Renaud Gaubert and Jiaying Zhang. 2017. Device manager proposal. Kubernetes design proposal. (July 2017). Retrieved 4 Mar. 2022 from https://github.com/kubernetes/design-proposals-archive/blob/main/resource-management/device-plugin.md.

[58] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*. ACM, 169–178. DOI: 10.1145/1536414.1536440.

[59] Craig Gentry and Shai Halevi. 2011. Implementing Gentry's fully-homomorphic encryption scheme. In *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings* (Lecture Notes in Computer Science). Vol. 6632. Springer, 129–148. DOI: 10.1007/978-3-642-20465-4_9.

[60] Andrew Gerrand. C? Go? Cgo! The Go Blog. (17 Mar. 2011). Retrieved 7 Mar. 2022 from https://go.dev/blog/cgo.

[61] David Goltzsche, Signe Rüsch, Manuel Nieke, Sébastien Vaucher, Nico Weichbrodt, Valerio Schiavoni, Pierre-Louis Aublin, Paolo Costa, Christof Fetzer, Pascal Felber, Peter R. Pietzuch and Rüdiger Kapitza. 2018. EndBox: scalable middlebox functions using client-side trusted execution. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*. IEEE Computer Society, 386–397. DOI: 10.1109/DSN.2018.00048.

[62] Google LLC. 2021. Google cloud computing containers service. (8 Nov. 2021). Retrieved 3 Mar. 2022 from https://cloud.google.com/compute/docs/containers/.

[63] Christian Göttel, Lars Nielsen, Niloofar Yazdani, Pascal Felber, Daniel E. Lucani and Valerio Schiavoni. 2020. Hermes: enabling energy-efficient IoT networks with generalized deduplication. In *DEBS '20: The 14th ACM International Conference on Distributed and Event-based Systems, Montreal, Quebec, Canada, July 13-17, 2020*. ACM, 133–136. DOI: 10.1145/3401025.3404098.

[64] Christian Göttel, Rafael Pires, Isabelly Rocha, Sébastien Vaucher, Pascal Felber, Marcelo Pasin and Valerio Schiavoni. 2018. Security, performance and energy trade-offs of hardware-assisted memory protection mechanisms. In *37th IEEE Symposium on Reliable Distributed Systems, SRDS 2018, Salvador, Brazil, October 2-5, 2018*. IEEE Computer Society, 133–142. DOI: 10.1109/SRDS.2018.00024.

[65] Franz Gregor, Wojciech Ozga, Sébastien Vaucher, Rafael Pires, Do Le Quoc, Sergei Arnautov, André Martin, Valerio Schiavoni, Pascal Felber and Christof Fetzer. 2020. Trust management as a service: enabling trusted execution in the face of byzantine stakeholders. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*. IEEE, 502–514. DOI: 10.1109/DSN48063.2020.00063.

[66] gRPC Authors. 2022. gRPC. A high performance, open source universal RPC framework. Retrieved 4 Mar. 2022 from https://grpc.io/.

[67] Jinyu Gu, Zhichao Hua, Yubin Xia, Haibo Chen, Binyu Zang, Haibing Guan and Jinming Li. 2017. Secure live migration of SGX enclaves on untrusted cloud. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*. IEEE Computer Society, 225–236. DOI: 10.1109/DSN.2017.37.

[68] Shay Gueron. 2016. Memory encryption for general-purpose processors. *IEEE Security & Privacy*, 14, 6, 54–62. DOI: 10.1109/MSP.2016.124.

[69] Linley Gwennap. 1997. P6 microcode can be patched. *Microprocessor Report*.

[70] Bo Han, Vijay Gopalakrishnan, Lusheng Ji and Seungjoon Lee. 2015. Network function virtualization: challenges and opportunities for innovations. *IEEE Commun. Mag.*, 53, 2, 90–97. DOI: 10.1109/MCOM.2015.7045396.

[71] [SW] Juhyeng Han, OpenSSL library for SGX application 2017. URL: https://github.com/sparkly9399/SGX-OpenSSL.

[72] Juhyeng Han, Seong Min Kim, Jaehyeong Ha and Dongsu Han. 2017. SGX-Box: enabling visibility on encrypted traffic using a secure middlebox module. In *Proceedings of the First Asia-Pacific Workshop on Networking, APNet 2017, Hong Kong, China, August 3-4, 2017*. ACM, 99–105. DOI: 10.1145/3106989.3106994.

[73] Alexis Hancock. 2021. We encrypted the web: 2021 year in review. Electronic Frontier Fondation, (27 Dec. 2021). Retrieved 10 Aug. 2022 from https://www.eff.org/deeplinks/2021/12/we-encrypted-web-2021-year-review.

[74] Guy Harris and Michael Richardson. 2022. PCAP Capture File Format. Internet-Draft draft-ietf-opsawg-pcap-01. Work in Progress. Internet Engineering Task Force, (29 July 2022). https://datatracker.ietf.org/doc/draft-ietf-opsawg-pcap/01/.

[75] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank and Michael Menth. 2021. A survey on data plane programming with P4: fundamentals, advances, and applied research. *CoRR*, abs/2101.10632. arXiv: 2101.10632.

[76] Hewlett Packard Enterprise Development LP. 2022. Aruba EdgeConnect SD-Branch. Retrieved 1 Sept. 2022 from https://www.arubanetworks.com/assets/ds/DS_SD-WAN.pdf.

[77] Hex Five Security, Inc. 2020. HEX-Five MultiZone security. (9 Jan. 2020). Retrieved 4 July 2022 from https://hex-five.com/wp-content/uploads/2020/01/multizone-datasheet-20200109.pdf.

[78] Ariya Hidayat, Ivan De Marino, Vitaly Slobodin, Zack Weinberg et al. 2018. PhantomJS - scriptable headless browser. (2018). https://phantomjs.org/.

[79] Yun-Wu Huang and Philip S. Yu. 1999. Adaptive query processing for time-series data. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 15-18, 1999*. ACM, 282–286. DOI: 10.1145/312129.318357.

[80] 2022. IEEE standard for Ethernet. *IEEE Std 802.3-2022 (Revision of IEEE Std 802.3-2018)*, 1–7025. DOI: 10.1109/IEEESTD.2022.9844436.

[81] InfiniBand Trade Association. 2020. *InfiniBand Architecture Specification. Volume 1*. Version 1.4. (7 Apr. 2020).

[82] InfluxData Inc. 2021. *InfluxDB 1.3 documentation*. Retrieved 4 Mar. 2022 from https://docs.influxdata.com/influxdb/v1.3/.

[83] InfluxData Inc. 2021. *InfluxDB 1.3 documentation*. Chap. Query language. Retrieved 4 Mar. 2022 from https://docs.influxdata.com/influxdb/v1.3/query_language/.

[84] [SW] Intel Corporation, Intel SGX for Linux Open Source Gold Release version 1.9, 4 July 2017. Retrieved 16 Aug. 2022 from URL: https://github.com/intel/linux-sgx/releases/tag/sgx_1.9.

[85] [SW] Intel Corporation, Intel SGX Linux driver. 2017. Retrieved 4 Mar. 2022 from URL: https://github.com/intel/linux-sgx-driver/tree/eb61a95.

[86] [SW] Intel Corporation, Intel SGX for Linux Open Source Gold Release version 2.6, 6 July 2019. Retrieved 4 July 2022 from URL: https://github.com/intel/linux-sgx/releases/tag/sgx_2.6.

[87] Intel Corporation. 2021. 3rd Gen Intel Xeon Scalable processors. Product Brief. Retrieved 15 Mar. 2022 from https://www.intel.com/content/dam/www/public/us/en/documents/a1171486-icelake-productbrief-updates-r1v2.pdf.

[88] Intel Corporation. 2021. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z*. Version 076. (Dec. 2021).

[89] Intel Corporation. 2021. Which Intel Ethernet network adapters support iWARP and RoCE v2? Retrieved 5 Sept. 2022 from https://www.intel.com/content/www/us/en/support/articles/000031905/ethernet-products/700-series-controllers-up-to-40gbe.html.

[90] Intel Corporation. 2022. *Intel SGX Software Installation Guide. For Linux OS*. Version 2.16. (28 Mar. 2022). https://download.01.org/intel-sgx/sgx-linux/2.16/docs/Intel_SGX_SW_Installation_Guide_for_Linux.pdf.

[91] Intel Corporation. 2022. Intel Tofino intelligent fabric processors. Retrieved 1 Sept. 2022 from https://www.intel.com/content/dam/www/central-libraries/us/en/documents/tofino-product-family-brochure.pdf.

[92] Intel Corporation. 2022. Memory Tiering: A New Approach to Solving Modern Data Challenges. White paper. Retrieved 2 Sept. 2022 from https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2022-04/optane-memory-tiering-white-paper.pdf.

[93] [SW] Intel Corporation, Open Tofino 30 May 2022. Retrieved 1 Sept. 2022 from URL: https://github.com/barefootnetworks/open-tofino.

[94] Intel Corporation. 2022. Strengthen enclave trust with attestation. Retrieved 26 Aug. 2022 from https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/attestation-services.html.

[95] International Telecommunication Union. 2021. *Measuring digital development. Facts and figures 2021*. ISBN: 978-92-61-35401-5.

[96] 2021. *ioctl(2). Linux Programmer's Manual*. (22 Mar. 2021). Retrieved 7 Mar. 2022 from https://man7.org/linux/man-pages/man2/ioctl.2.html.

[97] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amir Saman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao and Steven Swanson. 2019. Basic performance measurements of the Intel Optane DC persistent memory module. *CoRR*, abs/1903.05714. arXiv: 1903.05714.

[98] JetBrains s.r.o. 2022. Kotlin programming language. Retrieved 15 Sept. 2022 from https://kotlinlang.org/.

[99] Anuj Kalia, David G. Andersen and Michael Kaminsky. 2020. Challenges and solutions for fast remote persistent memory access. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*. ACM, 105–119. DOI: 10.1145/3419111.3421294.

[100] Daeyoun Kang, Tae Joon Jun, Dohyeun Kim, Jaewook Kim and Daeyoung Kim. 2017. ConVGPU: GPU management middleware in container based virtualized environment. In *2017 IEEE International Conference on Cluster Computing, CLUSTER 2017, Honolulu, HI, USA, September 5-8, 2017*. IEEE Computer Society, 301–309. DOI: 10.1109/CLUSTER.2017.17.

[101] Vishnu Kannan and Victor Marmol. 2015. Resource usage monitoring in Kubernetes. *Kubernetes blog*, (12 May 2015). Retrieved 4 Mar. 2022 from https://kubernetes.io/blog/2015/05/resource-usage-monitoring-kubernetes/.

[102] David Kaplan. 2017. Protecting VM Register State with SEV-ES. White paper. (17 Feb. 2017). Retrieved 13 June 2022 from https://www.amd.com/system/files/TechDocs/Protecting%20VM%20Register%20State%20with%20SEV-ES.pdf.

[103] David Kaplan, Jeremy Powell and Tom Woller. 2016. AMD Memory Encryption. White paper. (21 Apr. 2016). Retrieved 13 June 2022 from https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf.

[104] Thomas Karagiannis, Richard Mortier and Antony I. T. Rowstron. 2008. Network exception handlers: host-network control in enterprise networks. In *Proceedings of the ACM SIGCOMM 2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Seattle, WA, USA, August 17-22, 2008*. ACM, 123–134. DOI: 10.1145/1402958.1402973.

[105] Daehyeok Kim, Yibo Zhu, Changhoon Kim, Jeongkeun Lee and Srinivasan Seshan. 2018. Generic external memory for switch data planes. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks, HotNets 2018, Redmond, WA, USA, November 15-16, 2018*. ACM, 1–7. DOI: 10.1145/3286062.3286063.

[106] [SW] Colin Ian King, Stress-ng. 2022. URL: https://github.com/ColinIanKing/stress-ng.

[107] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz and Yuval Yarom. 2019. Spectre attacks: exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 1–19. DOI: 10.1109/SP.2019.00002.

[108] Eddie Kohler, Robert Tappan Morris, Benjie Chen, John Jannotti and M. Frans Kaashoek. 2000. The Click modular router. *ACM Trans. Comput. Syst.*, 18, 3, 263–297. DOI: 10.1145/354871.354874.

[109] Diego Kreutz, Fernando M. V. Ramos, Paulo Jorge Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky and Steve Uhlig. 2015. Software-defined networking: A comprehensive survey. *Proc. IEEE*, 103, 1, 14–76. DOI: 10.1109/JPROC.2014.2371999.

[110] The Kubernetes Authors. 2021. *Kubernetes Documentation*. (7 Oct. 2021). Chap. Device Plugins. Retrieved 4 Mar. 2022 from https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/.

[111] The Kubernetes Authors. 2022. *Kubernetes Documentation*. Retrieved 3 Mar. 2022 from https://kubernetes.io/docs/home/.

[112] The Kubernetes Authors. 2022. *Kubernetes Documentation*. Chap. DeamonSet. Retrieved 7 Mar. 2022 from https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/.

[113] Dmitrii Kuvaiskii, Somnath Chakrabarti and Mona Vij. 2018. Snort intrusion detection system with Intel software guard extension (Intel SGX). *CoRR*, abs/1802.00508. arXiv: 1802.00508.

[114] Leslie Lamport. 2002. Paxos made simple, fast, and byzantine. In *Procedings of the 6th International Conference on Principles of Distributed Systems. OPODIS 2002, Reims, France, December 11-13, 2002* (Studia Informatica Universalis). Vol. 3. Suger, Saint-Denis, rue Catulienne, France, 7–9.

[115] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy and Zhi Liu. 2016. Embark: securely outsourcing middleboxes to the cloud. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*. USENIX Association, 255–273.

[116] [SW] Lawrence Berkeley National Laboratory, iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool. Retrieved 14 July 2022 from URL: https://github.com/esnet/iperf.

[117] 2021. *ld.so(8). Linux Programmer's Manual.* (27 Aug. 2021). Retrieved 8 Sept. 2022 from https://man7.org/linux/man-pages/man8/ld.so.8.html.

[118] Kevin Leach, Fengwei Zhang and Westley Weimer. 2017. Scotch: combining software guard extensions and system management mode to monitor cloud resource usage. In *Research in Attacks, Intrusions, and Defenses - 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18-20, 2017, Proceedings* (Lecture Notes in Computer Science). Vol. 10453. Springer, 403–424. DOI: 10.1007/978-3-319-66332-6_18.

[119] Kevin Lepak, Gerry Talbot, Sean White, Noah Beck and Sam Naffziger. The next generation AMD enterprise server product architecture. Hot Chips 29, (Aug. 2017). Retrieved 13 June 2022 from https://old.hotchips.org/wp-content/uploads/hc_archives/hc29/HC29.22-Tuesday-Pub/HC29.22.90-Server-Pub/HC29.22.921-EPYC-Lepak-AMD-v2.pdf.

[120] LF Projects, LLC. 2022. Data plane development kit. Retrieved 31 Aug. 2022 from https://www.dpdk.org/.

[121] Jingwei Li, Chuan Qin, Patrick P. C. Lee and Jin Li. 2016. Rekeying for encrypted deduplication storage. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*. IEEE Computer Society, 618–629. DOI: 10.1109/DSN.2016.62.

[122] Benoît Libert, Kenneth G. Paterson and Elizabeth A. Quaglia. 2012. Anonymous broadcast encryption: adaptive security and efficient constructions in the standard model. In *Public Key Cryptography - PKC 2012 - 15th International Conference on Practice and Theory in Public Key Cryptography, Darmstadt, Germany, May 21-23, 2012. Proceedings* (Lecture Notes in Computer Science). Vol. 7293. Springer, 206–224. DOI: 10.1007/978-3-642-30057-8_13.

[123] [SW] Niels Lohmann et al., JSON for Modern C++. 2018. URL: https://github.com/nlohmann/json.

[124] Youyou Lu, Jiwu Shu, Youmin Chen and Tao Li. 2017. Octopus: an RDMA-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. USENIX Association, 773–785. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu.

[125] Philip D. MacKenzie, Michael K. Reiter and Ke Yang. 2004. Alternatives to non-malleability: definitions, constructions, and applications (extended abstract). In *Theory of Cryptography, First Theory of Cryptography Conference, TCC 2004, Cambridge, MA, USA, February 19-21, 2004, Proceedings* (Lecture Notes in Computer Science). Vol. 2951. Springer, 171–190. DOI: 10.1007/978-3-540-24638-1_10.

[126] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd and Carlos V. Rozas. 2016. 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@ICSA 2016, Seoul, Republic of Korea, June 18, 2016*. ACM, 10:1–10:9. DOI: 10.1145/2948618.2954331.

[127] Ashley Melidosian. 2020. Using a Virtual Data Room for an M&A Transaction. White paper. Retrieved 22 Aug. 2022 from https://learnings.idealsvdr.com/app/uploads/2021/12/using-vdr-m-a-2020.pdf.

[128] Paul Menage, Paul Jackson and Christoph Lameter. Cgroups. (2004). Retrieved 7 Mar. 2022 from https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[129] MinIO, Inc. 2022. MinIO quickstart guide. Retrieved 23 Aug. 2022 from https://docs.min.io/.

[130] MongoDB Inc. 2018. *MongoDB C Driver*. Version 1.12.0. Retrieved 23 Aug. 2022 from https://mongoc.org/libmongoc/1.12.0/index.html.

[131] MongoDB Inc. 2019. *The MongoDB Manual*. Version 4.0. Retrieved 23 Aug. 2022 from https://www.mongodb.com/docs/v4.0/.

[132] Mathias Morbitzer, Manuel Huber, Julian Horsch and Sascha Wessel. 2018. SEVered: subverting AMD's virtual machine encryption. In *Proceedings of the 11th European Workshop on Systems Security, EuroSec@EuroSys 2018, Porto, Portugal, April 23, 2018*. ACM, 1:1–1:6. DOI: 10.1145/3193111.3193112.

[133] Nafiseh Moti, Frederic Schimmelpfennig, Reza Salkhordeh, David Klopp, Toni Cortes, Ulrich Rückert and André Brinkmann. 2021. Simurgh: a fully decentralized and secure NVMM user space file system. In *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021*. ACM, 46:1–46:14. DOI: 10.1145/3458817.3476180.

[134] David Naylor, Richard Li, Christos Gkantsidis, Thomas Karagiannis and Peter Steenkiste. 2017. And then there were more: secure communication for more than two parties. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2017, Incheon, Republic of Korea, December 12 - 15, 2017*. ACM, 88–100. DOI: 10.1145/3143361.3143383.

[135] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R. López, Konstantina Papagiannaki, Pablo Rodríguez Rodríguez and Peter Steenkiste. 2015. Multi-context TLS (mcTLS): enabling secure in-network functionality in TLS. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*. ACM, 199–212. DOI: 10.1145/2785956.2787482.

[136] Jacob Nelson. 2020. SwitchML RDMA example client. Microsoft Corporation, (2020). Retrieved 7 Sept. 2022 from https://github.com/microsoft/SwitchML/blob/main/RDMAExampleClient/README.md.

[137] Hai Nguyen and Vinod Ganapathy. 2017. EnGarde: mutually-trusted inspection of SGX enclaves. In *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*. IEEE Computer Society, 2458–2465. DOI: 10.1109/ICDCS.2017.35.

[138] NVIDIA Corporation. 2021. *MLNX_OFED Documentation*. Version 5.5. Chap. InfiniBand Fabric Utilities. Retrieved 18 Mar. 2022 from https://docs.nvidia.com/networking/display/MLNXOFEDv551032/InfiniBand+Fabric+Utilities.

[139] NVIDIA Corporation. 2022. ConnectX SmartNICs. Retrieved 5 Sept. 2022 from https://www.nvidia.com/en-us/networking/ethernet-adapters/.

[140] Meni Orenbach, Pavel Lifshits, Marina Minkin and Mark Silberstein. 2017. Eleos: exitless OS services for SGX enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*. ACM, 238–253. DOI: 10.1145/3064176.3064219.

[141] The P4.org Architecture Working Group. 2018. *$P4_{16}$ Portable Switch Architecture (PSA)*. Version 1.1. Retrieved 1 Sept. 2022 from https://p4.org/p4-spec/docs/PSA-v1.1.0.pdf.

[142] Tuomas Pelkonen, Scott Franklin, Paul Cavallaro, Qi Huang, Justin Meza, Justin Teller and Kaushik Veeraraghavan. 2015. Gorilla: a fast, scalable, in-memory time series database. *Proc. VLDB Endow.*, 8, 12, 1816–1827. DOI: 10.14778/2824032.2824078.

[143] Rafael Pires, David Goltzsche, Sonia Ben Mokhtar, Sara Bouchenak, Antoine Boutet, Pascal Felber, Rüdiger Kapitza, Marcelo Pasin and Valerio Schiavoni. 2018. CYCLOSA: decentralizing private web search through SGX-based browser extensions. In *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*. IEEE Computer Society, 467–477. DOI: 10.1109/ICDCS.2018.00053.

[144] Rafael Pires, Marcelo Pasin, Pascal Felber and Christof Fetzer. 2016. Secure content-based routing using Intel software guard extensions. In *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016*. ACM, 10. DOI: 10.1145/2988336.

[145] Raluca Ada Popa, Jacob R. Lorch, David Molnar, Helen J. Wang and Li Zhuang. 2011. Enabling security in cloud storage SLAs with CloudProof. In *2011 USENIX Annual Technical Conference, Portland, OR, USA, June 15-17, 2011*. USENIX Association. https://www.usenix.org/conference/usenixatc11/enabling-security-cloud-storage-slas-cloudproof.

[146] Jon Postel, ed. 1981. Internet protocol. RFC 791. (Sept. 1981). DOI: 10.17487/RFC0791.

[147] Liam Proven. 2022. Why the end of Optane is bad news for all IT. *The Register*, (1 Aug. 2022). Retrieved 2 Sept. 2022 from https://www.theregister.com/2022/08/01/optane_intel_cancellation/.

[148] Yi Qiao, Xiao Kong, Menghao Zhang, Yu Zhou, Mingwei Xu and Jun Bi. 2020. Towards in-network acceleration of erasure coding. In *SOSR '20: Symposium on SDN Research, San Jose, CA, USA, March 3, 2020*. ACM, 41–47. DOI: 10.1145/3373360.3380833.

[149] Irving S Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8, 2, 300–304. Retrieved 31 Aug. 2022 from https://www.jstor.org/stable/2098968.

[150] Charles Reiss, John Wilkes and Joseph L. Hellerstein. 2011. Google cluster-usage traces: format + schema. Tech. rep. Version 2.1. Google LLC, Mountain View, CA, USA, (Nov. 2011). https://github.com/google/cluster-data.

[151] Eric Rescorla. 2017. Update on TLS 1.3 middlebox issues. (6 Oct. 2017). Retrieved 16 Aug. 2022 from https://mailarchive.ietf.org/arch/msg/tls/yt4otPd5u_6fOzW02TEe2e-W5G0/.

[152] Martin Roesch. 1999. Snort: lightweight intrusion detection for networks. In *Proceedings of the 13th Conference on Systems Administration (LISA-99), Seattle, WA, USA, November 7-12, 1999*. USENIX, 229–238. http://www.usenix.org/publications/library/proceedings/lisa99/roesch.html.

[153] Paul Rösler, Christian Mainka and Jörg Schwenk. 2018. 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, London, United Kingdom, April 24-26, 2018*. IEEE, 415–429. DOI: 10.1109/EuroSP.2018.00036.

[154] Mark Russinovich. 2017. Introducing Azure confidential computing. *Microsoft Azure Blog*, (14 Sept. 2017). Retrieved 4 Mar. 2022 from https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/.

[155] Adi Shamir. 1979. How to share a secret. *Commun. ACM*, 22, 11, 612–613. DOI: 10.1145/359168.359176.

[156] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy and Vyas Sekar. 2012. Making middleboxes someone else's problem: network processing as a cloud service. In *ACM SIGCOMM 2012 Conference, SIGCOMM '12, Helsinki, Finland - August 13 - 17, 2012*. ACM, 13–24. DOI: 10.1145/2342356.2342359.

[157] Justine Sherry, Chang Lan, Raluca Ada Popa and Sylvia Ratnasamy. 2015. BlindBox: deep packet inspection over encrypted traffic. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*. ACM, 213–226. DOI: 10.1145/2785956.2787502.

[158] Aleksey Shipilev et al. 2022. JMH: Java microbenchmark harness. Oracle Corporation. Retrieved 23 Aug. 2022 from https://openjdk.org/projects/code-tools/jmh/.

[159] Manmeet Singh, Maninder Singh and Sanmeet Kaur. 10 days DNS network traffic from April-May, 2016. Version V2. Mendeley Data. DOI: 10.17632/zh3wnddzxy.2.

[160] Teja Singh, Sundar Rangarajan, Deepesh John, Carson Henrion, Shane Southard, Hugh McIntyre, Amy Novak, Stephen Kosonocky, Ravi Jotwani, Alex Schaefer, Edward Chang, Joshua Bell and Michael Co. 2017. Zen: a next-generation high-performance ×86 core. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. (Feb. 2017), 52–53. DOI: 10.1109/ISSCC.2017.7870256.

[161] João Soares, Carlos Goncalves, Bruno Parreira, Paulo Tavares, Jorge Carapinha, João Paulo Barraca, Rui L. Aguiar and Susana Sargento. 2015. Toward a telco cloud environment for service functions. *IEEE Commun. Mag.*, 53, 2, 98–106. DOI: 10.1109/MCOM.2015.7045397.

[162] Weibin Sun and Robert Ricci. 2013. Fast and flexible: parallel packet processing with GPUs and Click. In *Symposium on Architecture for Networking and Communications Systems, ANCS '13, San Jose, CA, USA, October 21-22, 2013*. IEEE Computer Society, 25–35. DOI: 10.1109/ANCS.2013.6665173.

[163] Igor Sysoev. 2022. nginx. Retrieved 24 Aug. 2022 from https://nginx.org/en/.

[164] Hassan Takabi, James B. D. Joshi and Gail-Joon Ahn. 2010. Security and privacy challenges in cloud computing environments. *IEEE Secur. Priv.*, 8, 6, 24–31. DOI: 10.1109/MSP.2010.186.

[165] 2021. *The GNU Privacy Guard Manual*. Version 2.3.3. (Oct. 2021). Retrieved 22 Aug. 2022 from https://www.gnupg.org/documentation/manuals/gnupg/.

[166] 2021. *The GNU Privacy Guard Manual*. Version 2.3.3. (Oct. 2021). Chap. Key related options. Retrieved 22 Aug. 2022 from https://www.gnupg.org/documentation/manuals/gnupg/GPG-Key-related-Options.html.

[167] Tian Tian and Chiu-Pi Shih. 2012. Software techniques for shared-cache multi-core systems. Intel Developer Zone. Intel Corporation, (2012). Retrieved 20 June 2022 from https://web.archive.org/web/20181013172838/https://software.intel.com/en-us/articles/software-techniques-for-shared-cache-multi-core-systems/.

[168] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia and Christof Fetzer. 2018. ShieldBox: secure middleboxes using shielded execution. In *Proceedings of the Symposium on SDN Research, SOSR 2018, Los Angeles, CA, USA, March 28-29, 2018*. ACM, 2:1–2:14. DOI: 10.1145/3185467.3185469.

[169] Chia-che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira and Donald E. Porter. 2014. Cooperation and security isolation of library oses for multi-process applications. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*. ACM, 9:1–9:14. DOI: 10.1145/2592798.2592812.

[170] Chia-che Tsai, Donald E. Porter and Mona Vij. 2017. Graphene-SGX: a practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. USENIX Association, 645–658. https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

[171] [SW] Sébastien Vaucher et al., Base container for applications using the official Intel SGX SDK. 2019. URL: https://github.com/sebva/docker-sgx.

[172] [SW] Sébastien Vaucher and Rafael Pires, SGX-aware container orchestrator. 2020. URL: https://github.com/sebva/sgx-orchestrator.

[173] Sébastien Vaucher, Rafael Pires, Pascal Felber, Marcelo Pasin, Valerio Schiavoni and Christof Fetzer. 2018. SGX-aware container orchestration for heterogeneous clusters. In *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*. IEEE Computer Society, 730–741. DOI: 10.1109/ICDCS.2018.00076.

[174] [SW] Sébastien Vaucher, Rafael Pires and Christof Fetzer, Modifed Intel SGX Linux driver. 2017. URL: https://github.com/sebva/linux-sgx-driver.

[175] Sébastien Vaucher, Valerio Schiavoni and Pascal Felber. 2019. Stress-SGX: load and stress your enclaves for fun and profit. In *Networked Systems - 6th International Conference, NETYS 2018, Essaouira, Morocco, May 9-11, 2018, Revised Selected Papers* (Lecture Notes in Computer Science). Vol. 11028. Springer, 358–363. DOI: 10.1007/978-3-030-05529-5_24.

[176] Sébastien Vaucher, Niloofar Yazdani, Pascal Felber, Daniel E. Lucani and Valerio Schiavoni. 2020. ZipLine: in-network compression at line speed. In *CoNEXT '20: The 16th International Conference on emerging Networking EXperiments and Technologies, Barcelona, Spain, December, 2020*. ACM, 399–405. DOI: 10.1145/3386367.3431302.

[177] Rasmus Vestergaard, Qi Zhang and Daniel E. Lucani. 2019. Lossless compression of time series data with generalized deduplication. In *2019 IEEE Global Communications Conference, GLOBECOM 2019, Waikoloa, HI, USA, December 9-13, 2019*. IEEE, 1–6. DOI: 10.1109/GLOBECOM38437.2019.9013957.

[178] Marko Vukolic. 2016. The quest for scalable blockchain fabric: proof-of-work vs. BFT replication. In *Open Problems in Network Security - IFIP WG 11.4 International Workshop, iNetSec 2015, Zurich, Switzerland, October 29, 2015, Revised Selected Papers* (Lecture Notes in Computer Science). Vol. 9591. Springer, 112–125. DOI: 10.1007/978-3-319-39028-4_9.

[179] Daryle Walker. 2021. *Boost Library Documentation*. Version 1.78.0. Chap. 12. Boost.CRC. Retrieved 17 Mar. 2022 from https://www.boost.org/doc/libs/1_78_0/doc/html/crc.html.

[180] Shie-Yuan Wang, Jun-Yi Li and Yi-Bing Lin. 2020. Aggregating and disaggregating packets with various sizes of payload in P4 switches at 100 Gbps line rate. *J. Netw. Comput. Appl.*, 165, 102676. DOI: 10.1016/j.jnca.2020.102676.

[181] Shie-Yuan Wang, Chia-Ming Wu, Yi-Bing Lin and Ching-Chun Huang. 2019. High-speed data-plane packet aggregation and disaggregation by P4 switches. *J. Netw. Comput. Appl.*, 142, 98–110. DOI: 10.1016/j.jnca.2019.05.008.

[182] John Wilkes. 2011. More Google cluster data. *Google research blog*, (Nov. 2011). Retrieved 4 Mar. 2022 from https://ai.googleblog.com/2011/11/more-google-cluster-data.html.

[183] XipLink Inc. 2019. XA optimization appliances overview. Retrieved 1 Sept. 2022 from https://web.archive.org/web/20220306055419/http://www.xiplink.com/media/XipLink-XA-Appliances-Overview.pdf.

[184] Niloofar Yazdani, Lars Nielsen and Daniel E. Lucani. 2020. Memory-aware online compression of CAN bus data for future vehicular systems. In *IEEE Global Communications Conference, GLOBECOM 2020, Virtual Event, Taiwan, December 7-11, 2020*. IEEE, 1–6. DOI: 10.1109/GLOBECOM42002.2020.9348074.

[185] Xingliang Yuan, Xinyu Wang, Jianxiong Lin and Cong Wang. 2016. Privacy-preserving deep packet inspection in outsourced middleboxes. In *35th Annual IEEE International Conference on Computer Communications, INFOCOM 2016, San Francisco, CA, USA, April 10-14, 2016*. IEEE, 1–9. DOI: 10.1109/INFOCOM.2016.7524526.

[186] Wei Zhang, Guyue Liu, Ali Mohammadkhan, Jinho Hwang, Kadangode K. Ramakrishnan and Timothy Wood. 2016. SDNFV: flexible and dynamic software defined control of an application-and flow-aware data plane. In *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016*. ACM, 2. DOI: 10.1145/2988336.

[187] Xiao Zhang, Yinrun Lyu, Yanjun Wu and Chen Zhao. 2018. MixHeter: A global scheduler for mixed workloads in heterogeneous environments. *J. Parallel Distributed Comput.*, 111, 93–103. DOI: 10.1016/j.jpdc.2017.07.007.

[188] Bohong Zhu, Youmin Chen, Qing Wang, Youyou Lu and Jiwu Shu. 2021. Octopus^{+}: an RDMA-enabled distributed persistent memory file system. *ACM Trans. Storage*, 17, 3, 19:1–19:25. DOI: 10.1145/3448418.

[189] Philip R Zimmermann. 1995. *The official PGP user's guide*. MIT press, Cambridge, MA, USA. ISBN: 978-0-262-74017-3.