# Secure Large-Scale Outsourced Services
# Founded on Trustworthy Code Executions

Bruno Vavala

CMU-CS-17-118
July 2017

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Faculty of Sciences
University of Lisbon
1749-16 Lisbon, Portugal

**Thesis Committee:**
Peter Steenkiste, Co-Chair
Nuno Neves, University of Lisbon, Co-Chair
Anupam Datta
Vyas Sekar
Antonia Lopes, University of Lisbon

*Submitted in partial fulfillment of the requirements for the degree of*
*Doctor of Philosophy in Computer Science*

This research was sponsored by Fundação para a Ciência e Tecnologia (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program under grant SFRH/BD/51562/2011 (until August 2016) and the Information and Communication Technology Institute at Carnegie Mellon University, by FCT through project UID/CEC/00408/2013 (LaSIGE), by the EC through project FP7-607109 (SEGRID) and project H2020-643964 (SUPERCLOUD). The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the Portuguese government, the U.S. government or any other entity.

**Keywords:** Trusted Computing, Cloud Security, Trusted Execution Abstraction, Execution Integrity, Code Identity, Large-scale Data, TPM, Intel SGX, Efficient Execution Verification, Passive Replication, Service Availability

This work was partially supported by the Fundação para a Ciência e Tecnologia (FCT) through research grant SFRH/BD/51562/2011 (until August 2016) and through project UID/CEC/00408/2013 (LaSIGE), by the Information and Communication Technology Institute at Carnegie Mellon University, by the EC through project FP7-607109 (SEGRID) and project H2020-643964 (SUPERCLOUD).

*A Mamma, Papà, Titti,*
*a tutta la mia Famiglia e a Lucia,*
*per il loro costante e incondizionato affetto.*

# Abstract

The Cloud Computing model has incentivized companies to outsource services to third-party providers. Service owners can use third-party computational, storage and network resources while avoiding the cost of acquiring an IT infrastructure. However, they have to rely on the trustworthiness of the third-party providers, who ultimately need to guarantee that the services run as intended.

The fundamental security challenge is how to empower companies that own and outsource such services, or clients using them, to check service execution on the remote cloud platform. A promising approach is based on hardware-enforced isolation and attestation of the service execution. Assuming that hardware attacks are infeasible, this protects the service from other malicious software or untrusted system administrators. Also, it allows clients to check that the results were produced as intended. While this paradigm is well known, previous work does not scale with large code and data sizes, lacks generality both with respect to hardware (e.g., either uses Trusted Platform Modules, TPMs, or Intel SGX) and software (e.g., only supports MapReduce applications), and makes undesirable security tradeoffs (e.g., resorts to a large Trusted Computing base, or TCB, to run unmodified services, or a small TCB but with limited functionality).

This thesis shows how to secure the execution of large-scale services efficiently and without these compromises. From the perspective of a client that sends a request and receives a response, trust can be established by verifying a small proof of correct execution that is attached to the result. On the remote provider's platform, a small trusted computing base enables the secure execution of generic services composed of a large source code base and/or working on large data sets, using an abstraction layer that is implementable on diverse trusted hardware architectures.

Our small TCB implements three orthogonal techniques that are the core contributions of this thesis. The first one targets the identification (and the execution) of only the part of code that is necessary to fulfill a client's request. This allows an increase both in security and efficiency by leaving any code that is not required to run the service outside the execution environment. The second contribution enables terabyte-scale data processing by means of a secure in-memory data handling mechanism. This allows a service to retrieve data that is validated on access and before use. Notably, data I/O is performed using virtual memory mechanisms that do not require any system call from the trusted execution environment, thereby reducing the attack surface. The third contribution is a novel fully-passive secure replication scheme that is tolerant to software attacks. Fault-tolerance delivers availability guarantees to clients, while passive replication allows for computationally efficient processing. Interestingly, all of our techniques are based on the same abstraction layer of the trusted hardware. In addition, our implementation and experimental evaluation demonstrate the practicality of these approaches.

# Resumo

O modelo de computação baseado em Nuvem incentivou as empresas a externalizar serviços a fornecedores terceiros. Os proprietários destes serviços podem utilizar recursos externos de computação, armazenamento e rede, evitando o custo de aquisição de uma infraestrutura IT. No entanto, têm de confiar que os serviços de fornecedores terceiros funcionem como planeado.

O desafio fundamental da segurança é fazer com que as empresas que possuem e externalizam serviços, ou clientes que utilizam estes, possam controlar a execução do serviço na plataforma remota baseada em Nuvem. Uma abordagem promissora é o isolamento e a atestação da execução do serviço a nível hardware. Assumindo que os ataques ao hardware não são possíveis, o serviço fica protegido contra software malicioso ou administradores de sistema suspeitos. Além disso, permite aos clientes controlarem que os resultados tenham sido produzidos como planeado. Embora esta abordagem seja bem conhecida, os trabalhos anteriores não escalam com grandes quantidades de código e dados, carecem de generalidade em relação ao hardware (e.g., utilizam TPMs ou SGX) e ao software (e.g., recorrem a uma Trusted Computing base, ou TCB, complexa para a execução de serviços não modificados, ou a uma TCB simplificada que tem funcionalidades limitadas).

Esta tese propõe uma proteção para a execução de serviços de grande escala de forma eficiente e sem as limitações anteriores. Da perspectiva de um cliente que envia um pedido e recebe uma resposta, a confiança pode ser estabelecida através de uma pequena prova de que a execução foi correcta que é anexada à resposta. Na plataforma do fornecedor remoto, um pequeno dispositivo de computação fiável permite a execução segura de serviços genéricos constituídos por uma grande quantidade de código e/ou que processam grandes conjuntos de dados, utilizando um nível de abstração que pode ser implementado em diversas arquitecturas de hardware fiável.

A nossa TCB simplificada implementa três técnicas independentes que são os contributos centrais desta tese. A primeira foca-se na identificação (e na execução) apenas da parte de código que é precisa para completar um pedido de um cliente. Isto permite um aumento de segurança e eficiência porque o código que não é necessário para executar o serviço fica fora do ambiente de execução. A segunda contribuição permite o processamento de dados na escala de um terabyte através de um mecanismo seguro de gestão dos dados em memória. Isso permite a um serviço carregar dados que são validados quando são acedidos e antes de serem utilizados. Em particular, a inserção e a saída dos dados é feita utilizando mecanismos de memória virtual que não necessitam de chamadas de sistema a partir do ambiente de execução fiável, reduzindo portanto a superfície de ataque. A terceira contribuição é um novo esquema de replicação seguro completamente passivo que é tolerante ataques de software. A tolerância a faltas garante disponibilidade aos clientes, enquanto a replicação passiva permite um processamento eficiente do ponto de vista computacional. Curiosamente, todas as técnicas são baseadas no mesmo nível de abstração do hardware fiável. Além disso, a nossa implementação e avaliação experimental demonstram a praticidade destas abordagens.

## Riassunto

Il modello di calcolo basato su Cloud ha incentivato le aziende a esternalizzare servizi a fornitori terzi. Oggi i proprietari di servizi possono disporre di una grande varietà di risorse esterne per le loro esigenze di calcolo, di archiviazione e di rete, evitando così il costo di acquisizione di una infrastruttura IT. Devono, però, affidarsi a fornitori terzi i quali, a loro volta, hanno bisogno di garantire che i servizi funzionino come previsto.

La sfida fondamentale relativa alla sicurezza è come rendere aziende e clienti che usano risorse esterne in grado di controllare l'esecuzione del servizio sulla piattaforma Cloud remota. Un approccio promettente è basato sull'isolamento e attestazione a livello hardware dell'esecuzione del servizio. Partendo dal principio che attacchi hardware non possono accadere, il meccanismo protegge il servizio da altri programmi maliziosi o da amministratori di sistema inaffidabili. Inoltre, il meccanismo permette ai clienti di controllare che i risultati siano stati prodotti come pianificato. Sebbene tale approccio sia ben noto, i precedenti lavori di ricerca non operano in modo soddisfacente con grandi quantità di codice e dati, sono carenti di generalità sia con rispetto all'hardware (e.g., usano o TPMs o SGX) che al software (e.g., supportano solo applicazioni MapReduce), e fanno compromessi indesiderati sulla sicurezza (e.g., ricorrono ad una grande Trusted Computing base, o TCB, per eseguire servizi non modificati, oppure ad una piccola TCB ma con funzionalità limitate).

Questa tesi mostra come proteggere l'esecuzione di servizi di larga scala in modo efficiente e senza tali compromessi. Dalla prospettiva di un cliente che invia una richiesta e riceve una risposta, la fiducia può essere verificata con una piccola prova di corretta esecuzione in allegato alla risposta. Sulla piattaforma del fornitore remoto, una piccola base di calcolo affidabile permette l'esecuzione sicura di servizi generici, che possono avere anche un grande codice sorgente e lavorare su grandi insiemi di dati, utilizzando un livello di astrazione che può essere implementato su diverse architetture di hardware affidabile.

La nostra piccola TCB implementa tre tecniche indipendenti che sono i contributi centrali di questa tesi. La prima si concentra sull'identificazione (e l'esecuzione) della sola parte di codice che è necessaria per completare una richiesta di un cliente. Questo permette un miglioramento in termini di sicurezza e di efficienza poichè lascia fuori dall'ambiente di esecuzione ogni altro codice che non è necessario per il funzionamento del servizio. Il secondo contributo permette il processamento di dati nella scala di un terabyte attraverso un meccanismo sicuro di gestione dei dati in memoria. Tale meccanismo consente di caricare dati che vengono validati al momento dell'accesso e prima di essere utilizzati. In particolare, l'inserimento/uscita dei dati è svolto utilizzando meccanismi di memoria virtuale che non richiedono alcuna chiamata di sistema dall'ambiente di esecuzione affidabile, riducendo quindi la superficie di attacco. Il terzo contributo è un nuovo schema di replicazione sicuro, completamente passivo, che è tollerante ad attacchi software. La tolleranza ai guasti garantisce la disponibilità del servizio ai clienti, mentre la replicazione passiva permette un processamento efficiente dal punto di vista computazionale. Curiosamente, tutte le tecniche sono basate sullo stesso livello di astrazione dell'hardware affidabile. Inoltre, la nostra implementazione e la valutazione sperimentale dimostrano la praticità di questi approcci.

## Acknowledgments

After so much hard work, it is time to look back and recognize that I would not be sitting here writing these acknowledgments without the support of many people that helped me reaching this far.

First of all, I am deeply grateful to my advisors Professor Nuno Neves (at FCUL) and Professor Peter Steenkiste (at CMU). They fostered my personal and professional growth with care and patience (a lot!). All of our interactions helped me to discard weak and seemingly interesting ideas and to concentrate my research efforts towards innovative and ambitious goals. Each and every thoughtful feedback that I received from them stimulated me to better shape my ideas and encouraged me to improve my research work. A special and warm thank you goes to Nuno, who went far beyond his duties to support me in Portugal since before I decided to join also CMU.

I would like to say thank you to the faculties that accepted to join my thesis committee, namely Professor Anupam Datta, Professor Antonia Lopes and Professor Vyas Sekar. I have received valuable feedback from them for improving my thesis and its presentation. I hope there will be opportunities in the future to discuss and address together with them new questions and challenges that this thesis raises.

Many thanks to the amazing managers Pedro Gonçalves at LaSIGE and Deborah Cavlovich at CMU. Thank you for your promptness and efforts to help me in every administrative process I had to go through.

As my journey began in Rome, I would like to thank my Master's thesis advisor Professor Alessandro Mei, Professor Luigi Vincenzo Mancini and Professor Federico Massaioli, from Sapienza University of Rome, for encouraging me to pursue academic research. Also, I would like to give a big hug to Dora (Bionda) Spenza, Marco (Ocram) Barbera, Julinda Stefa, Ornela (Contessa) Dardha, Blerina Sinaimeri, Alessandro (Zio) Cammarano, Antonio (Don) Davoli, Andrea (All-in) Cerone, Claudiu (Pam) Perta, Marco (Senza) Cortina. I believe the awesome time we spent together is part of this journey, and I am very happy that I could meet many of you in Lisbon.

My journey then continued in Portugal, and I am impressed by how many people impacted my professional and personal life. I would like to thank (with my great great pleasure) Professor Paulo Verissímo who hosted me at LaSIGE in the early days. Also, many thanks to Professors Alysson Bessani, Fernando Ramos, Antonio Casimiro, Miguel Correia and Iberia Meideros for the insightful discussions, for the feedback they gave me on papers and presentations and for the help in organizing the research group meetings. A big thank you also goes to Monica and Rudra (o Indiano) Dixit, Juliana (Vovó) Veronese, João (mari...) Antunes, Vinicius Cogo, Je-

ferson (moleque) Souza, Patricia Gonçalves. Right after I moved to a new country, they made me feel at home. And similarly did the friends and colleagues that I met later, namely André (eu sou o maior) Nogueira, Henrique Moniz, Miguel (Presidente) Garcia, Leticia Fleig, Pedro (MapReduce) Costa, Luís (Crypto) Brandão, Diego (SDN) Kreutz, Tiago Oliveira, Ricardo Mendes, Tiago Cogumbreiro, Ricardo (Pato) Fonseca, João Sousa, Tulio Ribeiro, André (Mariachi) Santos, Simão Fontes and Morgana, Rui Fontes and Janete. Many thanks also to Marcia, Maria Clara and Leandro (o-irmão-do-Rudra) for introducing me to the warmth and the delicacies of the Brazilian culture in their house in Montijo. A special thank you to Vinicius Cogo for lots and lots of support with the Quinta testbed, and to André Nogueira for patiently discussing low-level implementation details; my experiments would not have been possible without them. A big hug also goes to Vincenzo Rocca who helped me since the very first day in Lisbon, while a big "beijinho" goes to my roommates Sónia Barbosa, Cristina Oliveira, Teresa Matos, Debora Sanches, who felt all of the good and the bad things of living in a house with a PhD student.

And my journey then continued at CMU. I am very grateful for all the good time spent in Pittsburgh with Cristian (mamma mia) Cassella, Sarah Loos and Jeremy (yes, I owe you a glass of wine!), Joana and Miguel Araujo (obrigado pelo bolo), Diana and João Martins (sim, eu sou boa pessoa, mas não danço), Giorgio De Pieri (a Veneto lost in Pittsburgh), Enico Iaia (who saved my life in the Death Valley), Jeronimo Segovia (from eh-Spain), Sid (always-happy) Ghosh; my soccer teammates Hugo Pinto, Matteo Rinaldi, Hugo Gonçalves and Philipp Reisinger; Sophie (beautiful dresses) and Matt Mukerjee, David Naylor (The Graphic Designer), John (One-day-I'll-know-how-to-play-foosball) Wright, Yuchen Wu, George Nychis, Stefan Muller (The Actor), Debjani and Wolf Richter (dude, I do not forget, and Deb neither!).

I want to conclude with a big big thank you to my beautiful family for always supporting me relentlessly. Thank you Mamma, Papà, Titty and Lucia. Luckily for me you are always there when I succeed, when I need some advice or a few words of comfort. Thank you also to my dear cousins in Canada, who hosted me for Christmas. With all of you, Toronto could not look any cozier to me. And thank you also to Vincenzo Rocca, Magda, Clara and (I-do-what-I-want-)Francesca, who made me feel as part of their family in Portugal.

Certainly, I have accidentally omitted somebody. If you believe you are missing here, you probably are (sorry about that!) and, to recover from my mistake, I am more than willing to give you a big hug in person instead.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| AES | Advanced Encryption Standard |
| AEX | Asynchronous EXit |
| AIK | Attestation Identity Key |
| AR | Active Replication |
| BFT | Byzantine Fault Tolerant |
| CA | Certification Authority |
| DAA | Direct Anonymous Attestation |
| EPC | Enclave Page Cache |
| EPID | Enhanced Privacy ID |
| EPT | (Intel) Extended Page Tables |
| fvTE | flexible and verifiable Trusted Execution (protocol) |
| I/O | Input/Output |
| IAS | Intel Attestation Service |
| IMEL | In-Memory Embedded Locator |
| ISV | Independent Software Vendor |
| IT | Information Technology |
| LaSt$^{\text{GT}}$ | LArge STate on a Generic Trusted component |
| KDF | Key Derivation Function |
| LPC | Low Pin Count (bus) |
| MAC | Message Authentication Code |
| NPT | (AMD) Nested Page Tables |
| PAL | Piece of Application Logic |
| PCR | Platform Configuration Register |
| PR | Passive Replication |
| PSE | Platform Specific Enclave |

| | |
|---|---|
| SDK | Software Development Kit |
| SEV | (AMD) Secure Encrypted Virtualization |
| SECS | SGX Enclave Control Structure |
| SGX | (Intel) Secure Guard eXtensions |
| SHA | Secure Hash Algorithm |
| SLoC | Source Lines of Code |
| SMR | State Machine Replication |
| SME | (AMD) Secure Memory Encryption |
| SMM | State Map Manager |
| SSA | State Save Area (frame) |
| SVM | (AMD) Secure Virtual Machine |
| TCB | Trusted Computing Base |
| TCC | Trusted Computing Component |
| TPM | Trusted Platform Module |
| $\mu$TPM | micro Trusted Platform Module |
| TCS | Thread Control Structure |
| TOCTOU | Time Of Check Time Of Use |
| TXT | (Intel) Trusted eXecution Technology |
| UTP | Untrusted Third Party |
| V-PR | Verified Passive Replication |
| VFS | Virtual File System |
| VM | Virtual Machine |
| VMM | Virtual Machine Monitor |

# Chapter 1

# Introduction

As our society becomes more connected and information is used in a more pervasive way, the systems are getting increasingly complex to process higher amounts of data. In just a few decades, advancements in computing systems made it possible to conclude hundreds of millions of financial transactions per day [211], connect billions of people [209, 210], rapidly and massively scale the IT infrastructure of many companies [212], for example to allow cost-effective genome sequencing technology [213].

As building large-scale systems requires lots of resources for hardware acquisition, software development and maintenance, it is often convenient to run the services on third-party computational resources. Today, this practice is implemented through the Cloud Computing model. Several cloud providers (e.g., Amazon EC2 [1], Rackspace [2], Microsoft Azure [3], IBM Cloud [4]) offer scalable resources to service providers, who exploit them to run disparate service applications, such as clinical decision support [5], predictive risk assessment for diseases [6], malware and fraud detection [7, 8], sensitive financial accounting and business optimization [9, 223, 224] and genome analytics [10, 11].

The practice of outsourcing services however has a security drawback. The service provider does not physically own the hardware that runs the service application. This prevents the service provider from being able to check and secure both the hardware and the software (e.g., the OS) running it. In fact, the service provider has very little control over the service execution.

## 1.1 Security Implications of Code and Data Outsourcing

Due to a lack of physical ownership and control of computing resources, outsourced applications lack strong integrity guarantees [9, 12, 13, 14] for code and data. Also, as these resources are remotely accessed [214, 215, 216], the remote activity is difficult to monitor. As a consequence, new threats emerge that can tamper with an outsourced service execution.

Unfortunately, although guaranteeing security is a core competence of cloud providers, this is not enough to rule out real threats. For example, one third of the top threats listed by the Cloud Security Alliance [196] enable an attacker to tamper with the integrity of the computation and for the data in the cloud. Namely: (i) service hijacking [203, 204], (ii) malicious insiders [15], (iii) system vulnerabilities [205, 206] and (iv) shared technology issues [16, 17, 18]. These threats raise suspicions about the trustworthiness of the results produced by an outsourced service, and they can undermine the reputation of both cloud providers and service providers.

## 1.2 Alternatives for Securing Outsourced Services

Service providers and clients have little or no means to check whether the service has been correctly executed on the cloud and that the delivered results are valid. A few techniques are available to deal with these issues, namely: service re-execution, trusted auditors, secure computation, and trusted hardware.

Service re-execution means making the clients (or whoever receives the results of the service) re-execute the service code locally so to check whether the received results match their expectations [19]. Although this allows checking the correctness of the remote computation, it can be classified as unrealistic for multiple reasons. First, it requires the clients to have the code and enough resources to do the computation themselves. This invalidates the original motivation for outsourcing a service, since verification is as expensive as the remote execution. Second, it requires two executions, so it is inefficient. Third, two executions giving the same result must likely be deterministic, thereby ruling out several services that use sources of non-determinism (e.g., random values, thread schedulers).

Trusted auditors are essentially undercover clients that use the remote service with the sole intent of checking the results it delivers [20]. This method still involves some re-execution, though true clients do not have to run an expensive verification procedure. However, such a method is fundamentally probabilistic, hoping that any long-lasting malicious activity will eventually be detected by trusted auditors. Also, it is based on the strong assumption that an untrusted service provider is unable to identify the trusted auditors, so to behave correctly (only) with them.

Trusted hardware means leveraging a hardware root of trust inside the cloud provider's platform [21, 22] so to get security guarantees about the executed service—much like a 1978 proposal [23] advocated using physically secure hardware for privacy-preserving computation. In particular, hardware-based isolation mechanisms and cryptographic protocols can be used to secure service execution and to establish a secure channel between the client and the service execution. This allows the client to receive and verify a proof that the intended code was run and the intended data was processed. Such technique makes the assumption that the hardware and its manufacturer can be trusted and that the cryptographic protocols are secure. Also, the hardware should be available on the remote execution platform.

Secure computation finally suggests to cryptographically encode the service—for instance describing it as a boolean circuit and then using a circuit garbling technique to get confidentiality and authenticity guarantees [24]—and the input data so that the cloud provider can perform the computation and deliver an encoded output that a client can easily verify. At a high level, the verifiable computation scheme [25] works as follows. A client encodes a target function and the input, generating public and private values. Public values are provided to the untrusted cloud provider to produce an encoded output, while the client uses the private values to decode and verify the validity of the encoded output. Hence, this technique only makes cryptographic assumptions about the existence of computationally intractable problems, and it does not require trusted hardware. However, despite significant recent advances, protocols for secure and verifiable computation remain inefficient and difficult to apply to existing services.

## 1.3 Focusing on Hardware-based Secure Foundations

In this thesis, our objective is to deliver integrity guarantees for code and data to clients of outsourced services by making such services behave as expected and by empowering clients to check the results. Namely, when clients use a remote service and receive data from it, they should be able to easily make a trust decision: whether or not to accept the data as trustworthy.

This rules out the alternative of using trusted auditors, as "each" client should be empowered with such capability. In that case, in fact, clients rely on the auditors to monitor the intended behavior of the remote service and to notify the service providers if misbehavior is detected. So all clients simply implement a trivial "accept-all-data" policy, and do not make any effort for checking what they receive.

Making the trust decision "easily" also rules out service re-executions. As outsourced services benefit from, and may require, the significant computational resources of cloud providers, it is unlikely that clients using a low-power device could repeat the computation in a short time. In addition, service providers might not be willing to share their proprietary service code or data with their clients.

Secure Computation [26, 27, 28, 29, 30, 31, 32] is very attractive, but current solutions have severe limitations. On one hand, secure computation does not require making trust assumptions about the availability of trusted hardware and of a trusted manufacturer. On the other hand, however, secure computations are still orders of magnitude slower than the original baseline counterparts. Also, they are typically applied to relatively simple and small functions such as matrix multiplication, SHA family of cryptographic hash functions, substring search, dot product and AES encryption algorithms.

In contrast, lots of progress has been made on trusted hardware recently [33]. First of all, several hardware architectures are being commercialized today. Notable examples are: TPMs [34], crypto-cards [35] and CPUs with an extended instruction set [189, 200]. Also, these have been used to secure, for instance, database applications [36, 37], unmodified applications [38], MapReduce applications [39], containers [40], data analytics [41], with a reasonably small overhead.

Clients can easily leverage such trusted hardware to get security guarantees with just a "hint". The hint is given by the trusted service providers to the clients, and it is the identity of a service code they should expect results from. Alternatively, it could also be a verification function [42] for the service identity. This identity is a secure cryptographic hash, and therefore it is hard to find two different codes that have the same identity. Similarly, the trusted hardware on the remote untrusted platform can attest the identity of the executed code and what data it processed. If clients can match the trusted and the attested identities, then they can establish trust in the result, or refuse to accept it otherwise. As it is clear, clients are not required to run, or know anything about, the service code. They can check the results independently from the service complexity.

For these reasons, this thesis focuses on the Trusted Computing area. In particular, it contributes new protocols for hardware-based secure code execution. These protocols aim at advancing the state of the art for large-scale outsourced services in terms of security, efficiency and generality from a hardware and a software perspective.

### 1.3.1 Computing Model for Secure Outsourced Services

Trusted hardware enriches the computing model (Figure 1.1) with a new party, i.e., the hardware manufacturer, and with the additional assumption that the hardware does not suffer physical attacks. The trusted hardware is installed within the cloud provider's platform and contains embedded (or securely provisioned) keys, inaccessible to the cloud provider, to authenticate itself to a client at verification time. Similarly, service providers outsource the (execution of their) service software to the cloud provider. The hardware manufacturer and the service providers

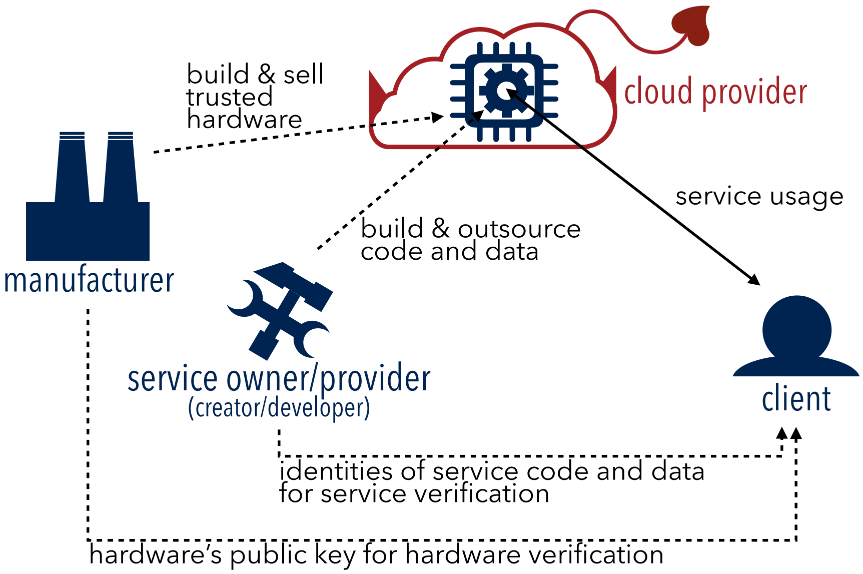


Figure 1.1: Model.

provide clients with the information that allows the secure verification of the service that is executed remotely at the cloud provider. Namely, the manufacturer tells the clients how to authenticate the results produced with the trusted hardware, while the service provider tells the clients what they should expect to be executed remotely i.e., service $f$ was executed over data $x$. So eventually clients can establish trust in the returned result $y$.

On the cloud provider's platform, the trusted hardware is able to set up an isolated environment for executing the service (see Section 1.3.4) that produces output data. In particular, the cloud provider feeds the hardware with the service code and the input data, and later retrieves the output data. Most importantly, the trusted hardware can sign the identities (hashes) of the executed service code and data with its keys. This ultimately gives any client the possibility to verify that the trusted hardware was used to execute the intended service. In addition, such a verification can be performed per-execution and thus at a very fine-grain level.

From a security perspective, the security guarantees about the service execution ultimately rely on the trusted manufacturer. In fact, the cloud provider is unable to tamper with the service execution while it is isolated. Execution isolation is in turn guaranteed by the trusted hardware, which inherits the trustworthiness of its manufacturer.

#### 1.3.1.1 System Model

Our basic model consists of a client that wants to verify the results received from a remote service possibly executing in an untrusted platform. The client relies on a trusted computing component (TCC) installed on the cloud provider's platform that provides a hardware root of trust. The TCC is able to sign messages with an embedded (or securely provisioned [194]) private key whose associated public key is certified by a Certification Authority (CA, possibly the manufacturer itself) that is trusted by the client. Also, we assume that the code developers and the data producers are additional entities, although in the rest of this thesis, unless otherwise specified, their roles are played for simplicity by the service provider. These entities are trusted by the client and provide information about the outsourced service code and the input data identities.

#### 1.3.1.2 Threat Model

We consider any cloud provider's platform fully untrusted except for the TCC. An adversary may take control of any software running at the cloud provider, including the OS. So the adversary is allowed to read and modify any data outside the TCC's trusted computing base (TCB). We assume the TCC is trustworthy due to its small hardware/software TCB, which does not include peripherals such as disk or network devices. Notice that the physical components to be trusted vary according to the TCC implementation. For example, considering TPM-based TCCs, physical components such as the CPU, the LPC bus, the memory modules and the communication bus have to be trusted. Instead, in an SGX-based TCC, only the CPU package has to be trusted—this also holds for other TCCs based on AMD SEV/SME [199, 200] or secure coprocessors [43, 193]. Physical attacks to these components are not considered.

The TCC must be able to provide an isolated execution environment and thus security guarantees for both code and data. The adversary is allowed to use the TCC using the TCC primitives (Section 2.3.1), for instance to run code in the isolated environment or to inject forged data into a trusted execution by modifying the input parameters. Also, after an isolated execution terminates, the adversary can tamper with any output data and attestation—which are transferred from the isolated environment to the untrusted environment.

Denial of Service (DoS) and cryptographic attacks are out of the scope of this thesis. DoS attacks are difficult to prevent since we already assume that the OS is untrusted and untrusted code can thus simply deny the use of the TCC. Cryptographic attacks are assumed to be computationally infeasible for the adversary's capabilities. In addition, side-channel attacks are not considered.

### 1.3.2 Available Hardware Architectures

A diverse set of trusted hardware technologies are commercially available today. Notable examples are: TPMs [34], crypto-cards [35] and CPUs with an extended instruction set (Intel SGX, AMD SEV) [189, 200].

TPMs are passive devices (i.e., they reply to the commands they receive) connected to a typical platform through the LPC bus—it allows low-bandwidth devices to communicate with the CPU. TPMs alone are not sufficient to enable trusted executions (Section 1.3.4). As they are passive devices, they need to work together with the main CPU using for instance AMD Secure Virtual Machine (SVM) or Intel Trusted Execution Technology (TXT).

Cryptographic coprocessor cards are high-performance hardware security modules. They relieve the main processor on the platform from the burden of running cryptographic operations (e.g., hashes and signatures) and instead execute them on a dedicated coprocessor inside a tamper-responding package—that deletes its internal private keys and certificates if its tamper-detection sensors are triggered. It is worth noting that these devices are designed to protect against physical tampering. This comes at a cost since they are typically at least two orders more expensive than TPMs. Also, they do not require SVM or TXT technology.

Finally, CPUs with an extended instruction set have been released recently. They provide dedicated CPU instructions for trusted executions, which are fully performed within the CPU package. They do not require external chips (as with TPMs) to deliver security guarantees and they are considered robust against physical and software attacks. Two examples are the Intel Secure Guard Extensions (SGX) [189] available on the Skylake microarchitecture, and AMD Secure Memory Encryption (SME) and Secure Encrypted Virtualization (SEV) [199, 200, 201] available on the Zen microarchitecture.

### 1.3.3 Can Such Hardware Be Actually Trusted?

Whether such hardware can be trusted or not is a debatable point and this thesis will not go deep into this discussion. Each architecture has different security strengths and weaknesses. For instance, while a TPM is a low-cost device that is considered tamper-resistant, communications on the LPC bus can be attacked [44], which is an issue if the end-points (CPU and TPM) are not mutually authenticated. Also, since there are multiple hardware components (TPM, CPU, memory), several manufacturers must be trusted.

As another example, although Intel SGX confines the security perimeter to the CPU package, some attacks are available to extract information from secure enclaves. Researchers have shown

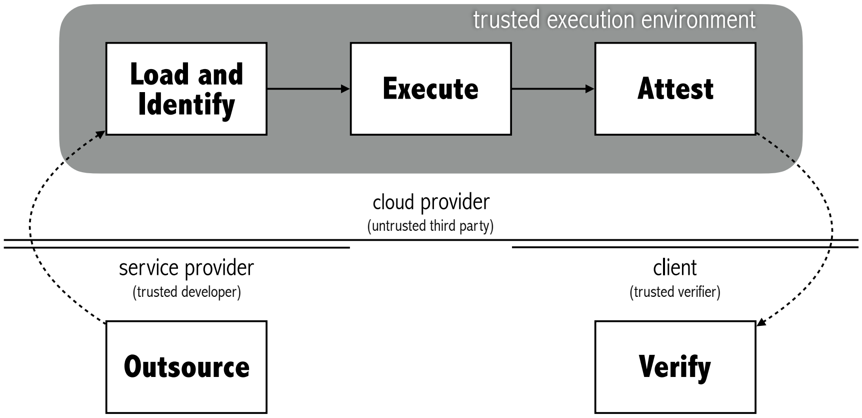


Figure 1.2: Anatomy of a trusted execution.

how to exploit synchronization bugs [45], or to implement side-channel attacks [46, 47, 48], and similarly how to mitigate them [49, 50, 51]. To the best of our knowledge however, no attacks have been shown that pose a threat to the integrity of the executed code.

To conclude, trusted hardware is being analyzed by the research community to discover flaws, and to improve its security and performance. The objective of this thesis is to build secure protocols on top of such hardware, though without assuming any specific hardware architecture.

### 1.3.4 Trusted and Verifiable Code Executions

The hardware architectures that we mentioned previously (Section 1.3.2) are necessary but not sufficient for running outsourced services. Since these services are user-level applications, low-level software that supports their trusted executions is necessary. Examples of systems that support such executions are: Flicker [52], TrustVisor [53], SICE [54], Fides [55], Haven [38], VC3 [39], Scone [40] and Graphene-SGX [56, 222]. Some are based on TPMs, some on SGX, while others like TrustedDB [36] (which also incorporates a database engine application) are based on the IBM 4764 cryptographic coprocessor. We will review these systems in Section 2.1.

A high-level picture of a trusted execution is displayed in Figure 1.2. The trusted service provider outsources the service code together with the input data, while the client possibly issues some requests. The service code is loaded and identified inside the trusted execution environment (i.e., completely isolated from other untrusted applications executing on the platform, including the OS). The service is then executed. Upon termination, the identity of the code, the input data and the results are attested using the trusted hardware's credentials and forwarded to the client. Attestation is a mechanism for securely reporting local platform state information (i.e., what code has been executed, what data has been processed, etc.) to a remote party (e.g., the client). The client (1) verifies the attestation, specifically that it was issued by trusted hardware certified by its manufacturer, and that (2) the hardware vouches for the received results, for the execution of the intended code and for the processed data.

The execution verification step should not be confused with "formal verification". The latter is about proving that the code behaves according to a formal specification [57, 58]. In a trusted execution, however, the attestation does not provide any information about the behavior of the code. Rather, it simply states that a well-identified code (whose identity is the hash of its binary, and is included in the attestation) has been loaded and executed and, similarly, the I/O data has been bound to that execution. Hence, formal verification refers to an orthogonal problem, out of the scope of this thesis, which can help to ensure that the executed code worked as it was originally intended, e.g., proving the absence of implementation bugs.

The cost of a trusted execution includes not just the service code execution, but also the cost of securing it (identification, attestation, etc.). A computationally expensive trusted execution can result in high perceived latency for the client to get the results, while a significant verification effort can translate into high cost for the client. In both cases, the trusted execution can end up being less attractive than a non-outsourced service execution, and therefore, these performance related aspects have to be taken into consideration while designing a solution.

### 1.3.5 The Ideal Theoretical and Practical Worlds

**In theory.** The following equations describe very concisely the concepts of a trusted service execution (right-side) which is the secure version of an generic service execution (left-side).

$$\underset{\text{(original execution)}}{y = F(x)} \qquad\qquad \underset{\text{(trusted execution)}}{(y, \pi) = T(F, x, n)}$$

Let us assume the service (code) we want to execute is described by a function $F$ that processes some input data $x$ and then delivers some output data $y$. The trusted execution of $F$ is given by another function $T$ that accepts in input the function $F$ (e.g., its binary code), the input data ($x$), a cryptographic nonce $n$ and eventually it delivers the output data ($y$) together with a verifiable proof of execution (or attestation) $\pi$. Such a proof is part of a message that includes the output $y$. The proof has the following format:

$$\pi =< id(F), id(x), id(y), n >_{K^-}$$

The function $id$ defines the identity of the input parameter (the code $F$, or the data $x$ or $y$) and it is defined as the cryptographic hash of the binary representation of the parameter. The message thus contains the identities of the function (our original service), the input and output data, and the nonce, and it is digitally signed with the hardware embedded private key $K^-$.

The recipient of $(y, \pi)$ can verify the execution as long as the function and input identities, the nonce and the public key $K^+$ associated to the hardware embedded private key are known. The execution verification is thus the authentication of a statement ($< \ldots >$) signed by the trusted hardware saying that code $F$ was executed in isolation over $x$ and returned $y$. A fresh nonce prevents untrusted parties from replaying such (trustworthy) statements, while a default public nonce explicitly provides such capability, thus allowing for example caching (and replaying) statements and outputs. It should be noticed that the parameters ($id(F)$, $id(x)$, $id(y)$, $n$, $K^+$, except the output $y$ itself) are small pieces of data. These can be given to a client directly and respectively by: the (service) function developers, the trusted (input) data source and the hardware manufacturer. The nonce is generated before the execution by the client.

We stick to this paradigm throughout the thesis, so the contributions described in the next chapters are heavily based on this theoretical definition of a trusted execution.

**In practice.** Ideally, it would be desirable to run trusted executions just as today's services are executed remotely. So let $F$ be the remote service (e.g., a web server, a database engine, some data analytics code), let $x$ be the outsourced data (e.g., a website data, a database, a data set), then perform the trusted execution to get the result (e.g., the dynamic content of the website, the output of a DB query, the analytics results) and finally make sure they are valid by verifying $\pi$.

## 1.4 Requirements and Challenges

In the real world however, several challenges must be addressed. For instance, building on the discussion in Section 1.3.5, we should design the function $T$, which enables trusted executions, in a way that is easy to use by service providers, generic enough to abstract (and be implementable on) several trusted hardware architectures, and whose implementation maintains the TCB small. Also, the implementation should allow the outsourced service $F$, the input $x$ and the output $y$ to scale efficiently to large code bases and large data sets, and to guarantee service availability to clients. In addition, the system should keep the proof of execution $\pi$ small and easy to verify.

In this thesis, we target four main requirements for securing outsourced services, which are related to hardware abstraction, execution integrity, scale and availability. In particular:

1. an abstraction for implementing generic services on diverse Trusted Computing architectures (Section 1.4.1);

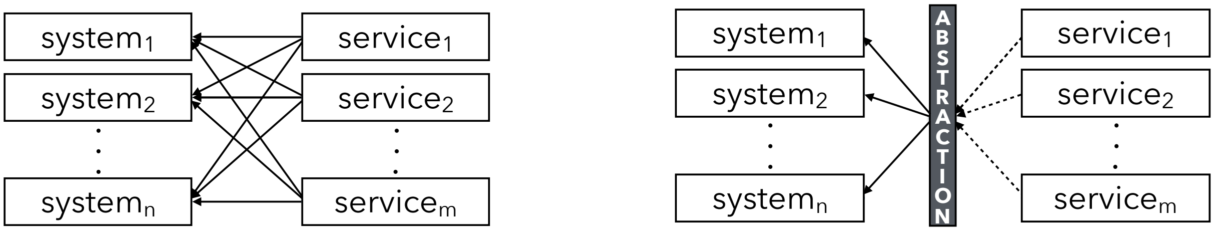


Figure 1.3: Possible system-service implementations without ($m \times n$, left-side) and with an abstraction ($m + n$, right-side). Arrows define an "implemented on" relation (e.g., $a \to b$ says that $a$ is implemented on $b$).

2. Integrity of large code (Section 1.4.2);
3. Ability to scale to large volumes of data (Section 1.4.3);
4. Service availability in spite of failures, such as crashes (Section 1.4.4).

Additional requirements that pervade our contributions are: execution efficiency and verification efficiency. Our requirements do not include supporting code or data confidentiality, which is left as a future extension of our work.

### 1.4.1 Abstracting the Hardware-based Secure Foundations

Given the parties involved in the system model (Section 1.3.1), we observe that the current practice of providing, developing on and using a specific Trusted Computing architecture is inconvenient. Today (Figure 1.3 left-side) $m$ service providers have to choose the Trusted Computing architecture they need, and implement their services for that specific architecture. Needless to say, supporting additional architectures quickly increases development and maintenance costs, due to the $n \times m$ different Trusted Computing architecture-service pairs that might result.

As several trusted hardware architectures and systems based on them (Section 1.3.4) offer similar capabilities, it is natural to ask whether service providers can build on an abstraction of such capabilities. The abstraction would be beneficial because it would provide a single intermediate between these systems and the services (Figure 1.3 right-side). In fact, each system developer could implement the abstraction, while each service provider could simply use the abstraction *"on account of what it does while completely disregarding how it works"* [59], thereby being *"able to ignore details not relevant to his application area, and to concentrate on solving his problem"* [60]. Although this slightly increases the cost for system developers—i.e., those who build a TPM-based trusted hypervisor or an SGX driver for trusted service executions, because they additionally have to extend these systems with the implementation of the abstraction—it simplifies service development, promotes comparisons among systems (e.g., the TPM-based trusted hypervisor or the SGX-based code that underlie the abstraction) and allows switching/upgrading hardware and system software. As a result, service providers could choose among multiple systems according to the security level of the trusted hardware (e.g., a low-cost TPM or an expensive tamper-responding cryptographic co-processor) and the delivered performance.

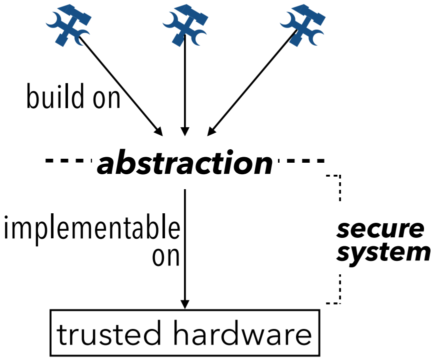


Figure 1.4: Usage and implementation of the trusted execution abstraction.

The challenge is coming up with such an abstraction (Figure 1.4), which can deliver a simple and small execution interface for service providers, and whose implementation is feasible on different TCC architectures. Borrowing from [61]:

- *Try to make your design be appropriate for hardware likely to appear in the future.*
- *By not relying on idiosyncratic features of the hardware, one makes porting to new platforms much easier.*

We point out that we do not aim at defining an abstraction that can be implemented on all available systems. For instance, TrustVisor and VC3 can hardly perform the same operations since the former focuses on self-contained x86 code, while the latter targets Hadoop applications.

We summarize our solution in Section 1.5.1 and discuss it in more detail in Chapter 2.

### 1.4.2 Integrity of Large Code

Reducing the TCB size is desirable [217] but maintaining the same functionality for a service is challenging. For example, purging the hard-disk out of the TCB means renouncing to a trusted mass storage support. As another example, although it is desirable to remove from the TCB large code bases such as libraries and the OS, which are the order of 1M [198] and 10M [197] lines of code respectively, this means giving up filesystems support and lots of other services. Whether (or how much of) these should be included in the TCB is today a design choice that has to find the right balance between security and generality. Many examples [38, 40, 62, 63] show that it is hard to reduce the TCB, because *"the interface between modern applications and operating systems is so complex"* [192].

The problem is exacerbated with large complex services that end up adding code to the TCB. As the executed code is meant to be remotely verified, and as a client verifies the identity of the code, such a large TCB has a significant impact on the code identification procedure.

On one hand, code identification provides the client with (a hash of the code and thus) integrity guarantees about the executed code. As these guarantees hold at load time, it cannot be ruled out that the execution is not compromised at run time, particularly in the case of a large complex code. Consequently, frequent code identification is desirable to refresh the integrity guarantees.

On the other hand, identifying a large amount of code can be expensive. Such cost is due to: the overhead of moving the code in and out the trusted execution environment, which depends on the used Trusted Computing architecture; the overhead of hashing the code, which depends on the used hash function; the size of the code. Solutions that attempt to reduce the first overhead would likely be architecture-specific. Instead, improving the efficiency of hash functions—which are already efficient—is difficult and out of scope. This leaves us with the challenge of how to reduce the service code in order to reduce the identification effort, so that the performance gain can finance the frequent identification. In the case of large code, we translate this challenge into how to identify just the required code when it has to be executed.

We summarize our solution in Section 1.5.2 and discuss it in more detail in Chapter 3.

### 1.4.3 Scaling to Large Volumes of Data

Outsourced services such as databases or data analytics typically work with large volumes of data that have to be verified. This allows a service to process correct data, as it was intended. Also, it allows the end-users to get integrity guarantees about the data.

In this scenario, we identify two main challenges, namely how to secure the data and how to supply the data to the executing service. Naive approaches such as hashing the data, or building a hash tree, or providing input data upfront are not satisfactory, because trusted executions environments, such as in XMHF-TrustVisor and SGX, have limited memory available. As the data scales up, it may not fit in memory; it is inefficient to make it available upfront to the service; hierarchical authentication data structures, such as hash trees, can be large as well and this must be taken into account. So memory is a concern.

Also, implementing a data I/O mechanism can have a negative impact on the TCB in three ways. Such a mechanism could enlarge the TCB with: 1) peripherals (e.g., the disk); 2) lots of code (e.g., the OS, or a filesystem) and 3) additional interface calls. So, the TCB size is a concern. The challenge is how to exclude additional hardware peripherals while maintaining a TCB with a small amount of code and a simple interface, while still being able to process large-scale data.

We summarize our solution in Section 1.5.3 and discuss it in more detail in Chapter 4.

### 1.4.4 Availability for Secure Outsourced Services

Availability is a requirement for both service providers and clients. Providers benefit from showing that their execution infrastructure is always-available, while clients have higher assurance that they will eventually receive the service results they requested. Recall (from Section 1.3.1.2) that we explicitly exclude DoS attacks from our threat model since availability trivially cannot be attained in the context of an untrusted cloud provider that mounts such an attack. A cloud provider's platform however could suffer network intrusions or crashes. So a highly available service should be able to provide the results despite such threats.

A replication scheme is the obvious solution, but both the active and passive replication alternatives have shortcomings. Next, we elaborate on the disadvantages of current replication schemes.

First, active replication, that is the replication of a service execution, can be computationally expensive. Also, the outcomes of the executions should match. This goal is difficult to achieve in complex non-deterministic systems, so service providers should either develop deterministic services or convert existing services to make them deterministic, which is a non-trivial task [64, 65, 66].

Alternatively, passive replication, that is the replication of a service's state (or data), is also hard to accomplish. In fact, any data that crosses the security perimeter of a trusted component should be protected. Data protection is based on cryptographic constructions that use secret keys that belong to a single trusted execution domain (i.e., the keys depend on the system and the trusted hardware that host the trusted execution). Therefore, replicating the data on different platforms simply makes difficult to validate (and thus to consider as trusted) the data in the trusted execution environment of another platform.

We summarize our solution in Section 1.5.4 and discuss it in more detail in Chapter 5.

## 1.5 Solving the Challenges for Secure Large-Scale Trusted Executions

In this thesis, we embrace these challenges and provide solutions for secure, efficient, scalable and available outsourced services. We show how to deliver these features for real world services that are composed of large code bases and use large data sets. Our contributions consist of three main complementary techniques: fresh integrity guarantees and performance for outsourced services with a large code base (Chapter 3); scalability to large amounts of data (Chapter 4); availability through cost-effective service replication (Chapter 5). An additional contribution is an abstraction for Trusted Computing on which the above techniques are built (Chapter 2).

Next we briefly provide the insights and an overview of our abstraction for trusted executions (Section 1.5.1) and of the techniques that, combined, enable to identify just the actively executed code (Section 1.5.2) for large-scale data processing (Section 1.5.3) with availability guarantees (Section 1.5.4).

### 1.5.1 An Abstraction for Trusted Executions

As Trusted Computing architectures are very diverse, it is natural to ask whether it is at all possible to define a single interface for a generic Trusted Computing Component (TCC). The key insight is that ultimately, all Trusted Computing architectures are used to implement a similar set of services needed to support a trusted execution. The TCC interface that we describe provides an abstraction for these services, rather than architectural features, allowing us to hide the differences between Trusted Computing architectures from application developers.

The abstraction for trusted code executions that we present is defined by a small set of primitives. These are our building blocks for the rest of the contributions. The primitives derive immediately from fundamental Trusted Computing concepts such as isolated execution, sealed storage, attestation, etc. They enable developers to build (or port) services on top of an abstract trusted component, to optimize the implementation, and require minimal effort and no knowledge of the details of the underlying trusted hardware architecture. In addition, the abstraction enables cloud providers to upgrade the trusted hardware (and/or the system built on top of it) as new architectures are released, or to compare these architectures with minimal effort. We discuss two implementations on two different Trusted Computing architectures.

### 1.5.2 Verifying an Application by Identifying just the Necessary Code

In order to avoid the TCC effort for loading and identifying unnecessary parts of a large sensitive service code, an obvious solution is to implement the service as a set of modules and to only load and identify the modules that are executed. However, a more significant challenge is how to efficiently verify that the control flow is respected, i.e., the intended modules are executed in the intended order.

The insight is to leverage mutually-authenticated sealed storage to chain the modules together and to let the client infer from a single attestation that the intended necessary set of modules were executed in the intended order. One attestation verification allows to establish trust in a trusted module. Since all modules are securely chained together pair-wise to exchange data, the verified module could only have exchanged data with another trusted module in the order defined by the secure chain.

The solution [67] that we present in Chapter 3 works as follows. First we split the code base into suitable code modules that can be composed together to implement the original code base logic. This can be achieved for example with known techniques such as program slicing [68] or program partitioning [69]. We then design a protocol that lets the Trusted Computing architecture load, identify and run only modules of the code base that are actually required during the execution. This allows us to reduce the active TCB and to save resources since the trusted component does not have to load and identify unneeded modules into the secure environment. The correct execution sequence of code modules is guaranteed by a robust and verifiable execution chain. Specifically, each module secures the application data using a secret key that depends on its own identity and the identity of the next module in the correct sequence. The protocol eventually allows the client to verify the execution chain efficiently by simply verifying a chain end-point to bootstrap trust in the whole chain. An interesting consequence of this construction is that the client does not require any knowledge of the exact execution order of the code modules, because the end-point verification ensures the correctness of the whole chain, whatever it may be for a specific execution.

### 1.5.3 Feeding Code with Large-Scale Data using Secure Virtual Memory Maps

Two insights are at the core of the contribution. In particular, (i) data I/O can be performed with no additional interface calls by simply handling page faults; (ii) memory constraints can be overcome by organizing the state into hierarchical components which can be mapped in memory efficiently and independently. Next, we elaborate on these insights.

First, by providing to the service code the entire view of the state in memory (i.e., code can access any data by simply walking the memory), we can reduce the data I/O problem to a virtual memory management problem. This can be solved without interface calls to the untrusted environment and without including OS code or peripherals in the TCB. Moreover, since we only assume the existence of virtual memory, which is a common feature in today's systems, the technique is applicable to different TCCs.

Second, mapping large-scale data in memory raises issues when the address space is small (e.g., 32-bit) or a limited set of addresses is a available (e.g., if the application can only access a small range of memory, say 1GB). When dealing with a terabyte of data, even the associated hash tree organized in an array may end up consuming lots of addresses. These issues can be solved by organizing the data, and the authentication metadata, into small components organized hierarchically and each one having its own authentication metadata. These components can be efficiently mapped in memory when required, and the data can be authenticated efficiently through the hierarchical data structure. Most importantly, components that are no longer needed in memory can be unloaded in order to allow reusing memory and addresses.

These insights led to the design and implementation of LaSt$^{GT}$ [41] which we describe in Chapter 4. LaSt$^{GT}$ can handle a **LA**rge **ST**ate on a **G**eneric **T**rusted component using a small TCB. The small TCB is the result of offloading many operations that are not security sensitive to untrusted code, thereby avoiding implementing and running them within the trusted execution environment. We specifically refer to data I/O to/from mass storage supports or through the network. As a result, LaSt$^{GT}$ does not include disks and network cards within the trust boundaries. LaSt$^{GT}$ simply lets untrusted code manage data I/O between the storage medium and the main memory and organize data into memory maps. Then, trusted code leverages paged virtual memory and memory maps to incrementally load the data that the service needs to process. The data is validated through a scalable authentication data structure before the service uses it, so to ensure that it does not process unintended data.

Interestingly, besides the trusted execution interface, LaSt$^{GT}$ does not require any additional system call or interface between the service application and the virtual memory management code. Our solution is purely based on virtual memory and thus generic and portable.

We point out however that there are two important differences with respect to VC3 [39]. First, the current design of LaSt$^{GT}$ does not target MapReduce applications but rather self-contained x86 applications, which are not supported in VC3. As an example, we will show how the SQLite database engine application [188] performs on LaSt$^{GT}$ for handling terabyte-scale databases. Second, LaSt$^{GT}$ currently does not focus on data/code confidentiality but only data integrity.

### 1.5.4 Improving Availability through Secure Replication

Passive replication is not robust because the execution at the primary is not replicated (only the state is replicated). The obvious insight is to leverage Trusted Computing to secure the primary execution, and to build a passive replication system inside the trusted environment of the replicas. This however comes with significant challenges such as reducing the TCB (e.g., storage and network peripherals) while preventing rollback attacks, and ensuring efficient operations for instance avoiding repeated attestation and verification steps.

These issues can be solved by leveraging trusted counters and sealed storage, which are common features provided by Trusted Computing architectures. Trusted counters prevent rollback attacks, as they can only be updated in a monotonically increasing fashion. Sealed storage allows a service to protect data so that it can be stored in untrusted storage. These mechanisms can be combined to implement a secure protocol, since the protocol's state can be securely stored while messages are exchanged among the replicas through the network.

This insight led to the concept of Verified Passive Replication (V-PR) [37] that we describe in Chapter 5. By leveraging a client's ability of verifying a trusted execution, V-PR avoids service re-executions and simply replicates the state of the primary replica to the backup replicas—i.e., any data modification performed by the primary replica is mirrored on the backup replicas (this is also known as Passive Replication, PR). Hence, V-PR faithfully follows the PR scheme rather than AR, but crucially enriches it with trusted executions to provide stronger security guarantees. V-PR is particularly attractive for computationally intensive services, since it does not require re-executions, and for the same reason it natively supports non-deterministic implementations. V-PR is also designed on our generic trusted execution primitives (Section 2.3) thereby inheriting all the benefits that such abstraction provides.

We conclude by emphasizing three interesting aspects of V-PR. First, it helps to provide low-cost replication for outsourced services. Second, it can benefits from our previous techniques (Section 1.5.2, Section 1.5.3) since it heavily leverages trusted executions, but it is not directly concerned with code identification or large-scale data processing. Third, V-PR is relevant to the Dependability community as it provides a novel hardware-based approach to secure and efficient replication that radically departs from the usual practice (i.e., AR).

## 1.6 Summary of Contributions

This thesis makes the following contributions:

1. We present a small set of primitives that provide an abstraction for trusted code executions. These are our building blocks for the rest of the contributions. We detail two implementations on two different Trusted Computing architectures, thereby showing that the abstraction can hide their details to developers.
2. We present a new approach for code identification. The technique allows loading and identifying within the trusted execution environment only the code that is necessary to compute the final output of an execution. The technique provides fresher code integrity guarantees, lower startup time, lower end-to-end latency and a reduced active TCB.
3. We describe a system that allows services to process a large state. The system leverages our set of primitives, and so generic trusted hardware, to guarantee data integrity and uses protocols purely based on virtual memory to supply data to the executing service. We implement the system by extending XMHF-TrustVisor, which was not designed for large-scale data processing. We evaluate the system with large-scale data applications and also show that it can outperforms the original hypervisor's implementation when services process a small state.
4. We show how to leverage trusted hardware to provide efficient service availability in a hybrid failure model—i.e., assuming that the TCC and any software running in the trusted execution environment can only fail by crashing, thereby excluding physical attacks to the TCC, while the rest of the system outside the TCC can experience arbitrary failures. Namely, we design an efficient and secure fully passive replicated system that avoids active service replication and also supports non-deterministic executions.
5. We implement our protocols on XMHF-TrustVisor and perform an experimental evaluation using real-world applications. The results show that our protocols are practical.

### Thesis Statement

We compress our findings into the following statement.

*Thesis Statement: " Outsourced large-scale services can be secured by means of an additional small trusted computing base, which can provide code and data integrity, and support efficient, scalable and available executions. Also, a handful of primitives can abstract the details of diverse trusted hardware architectures, thereby hiding their differences from application developers. "*

## Chapter 2

# Abstraction for Trustworthy Code Execution

This chapter is organized as follows.

- We first provide a brief introduction to Trusted Computing, describing how it began and evolved and what architectures it provides to us today (Section 2.1).
- We pick two different architectures, XMHF-TrustVisor and Intel SGX, and provide a brief description of how they work, thereby highlighting their differences (Section 2.2). These architectures serve as reference for the rest of the work.
- Given the variety of available architectures, we introduce our abstraction of a *trusted computing component* (TCC, Section 2.3). This allows us to devise general protocols and optimizations that work regardless of the specific details of each architecture. The abstraction is provided in terms of a small set of primitives (Section 2.3.1), for which we sketch a suitable implementation (Section 2.3.2) on both our reference architectures. We ultimately argue that their implementation can be adapted to additional architectures and future ones as well. These primitives are the base above which (i.e., by calling them), or below which (i.e., through a suitable implementation, as we highlight), we will describe the techniques that are the subject of the next chapters.

## 2.1 A Brief Introduction to Trusted Computing

The Trusted Computing area began with the proposal of a trusted open platform [21, 22] a bit more than a decade ago. The research work in the area is oriented at devising new architectures for a hardware and software root of trust, improving its performance and increasing its security. Such a root of trust is strongly bound to its hardware manufacturer—who makes the chip—and software developers—who make support (like a library) for applications—rather than the cloud provider who owns the hardware. Each architecture aims at providing a computational environment for general applications whose security ultimately relies on the root of trust. In particular, it provides strong isolation from the OS and other software on the platform, secret keys (based on the identity of the code) for protecting data in untrusted storage, and code identity attestation to enable remote software authentication. Next, we review previous work.

### 2.1.1 Binding Together Code and Data

From a systems perspective, the problem of checking a remote execution has been initially investigated in the context of mobile code. Originally it was informally stated as the Linking Problem (LP, between code and data) for mobile code executed on untrusted hosts [70, 71]. Namely, how a program executing at an untrusted host can sign the output of its computation, making sure that its signatures cannot be forged. It was already suggested that solving this problem would be difficult without resorting to special hardware [23, 70, 72], though researchers were also looking for solutions based on homomorphic encryption to avoid the need for this support.

Trusted hardware was thus recognized early on as an important building block to solve this problem, however, additional software is useful (if not necessary) to support actual code execution, for instance, by managing and protecting memory and providing I/O. BIND [73] solves the linking problem by exploiting the hardware-based isolated execution of a safety kernel, which guarantees the integrity of a critical piece of code through an attestation that a remote party can verify. Moreover, it highlights the importance of fresh integrity guarantees (i.e., attested measurements) because the computed identity of the executed code and data represents the application's state only at load time—code modifications due to attacks at execution time do not change the computed identity and can therefore do harm and go undetected. Unfortunately, BIND's security kernel was not implemented [74].

Flicker [74] bridges this gap providing the first implementation of a secure code execution architecture that minimizes the Trusted Computing Base (TCB). It works for small pieces of self-contained code and leverages the Late Launch technology [75, 76] to guarantee security. Late Launch provides the CPU extensions necessary to set up a dynamic root of trust (DRT). This refers to an isolated and measured execution environment that can be set up at any time by invoking a special security instruction.

A drawback of both BIND and Flicker is that they repeatedly use security features provided by a Trusted Platform Module (TPM). The consequence is a significant execution slowdown due to the interaction with a low-power TPM over a low-bandwidth LPC bus. This is particularly noticeable during code identification and attestation since hash computations and digital signatures are performed by the TPM.

### 2.1.2 Boosting Performance Leveraging Fast CPU

Given the slowness of the TPM, it has been seen as a mandatory requirement to run the bulk of code identifications and attestations on the fast CPU available on a platform. The solution is to perform the trusted execution of a small security module, that is identified and attested by the TPM; then such module enforces isolation and performs code identification and attestation of other software. In this way, the software to be protected leverages the security module and avoids direct interaction with the TPM. This paradigm has been implemented in systems like: TrustVisor [53], which improves performance; SICE [54], which further reduces the TCB; and Fides [55], which eases application development.

### 2.1.3 Reducing the Physical Trust Boundaries to a Single Chip

For several years, a security issue remained to be addressed, namely: how to confine the trust assumptions to a single chip. In fact, in typical TPM-based trusted executions, programs use the TPM as a hardware root-of-trust, the CPU for computing, and (unencrypted) main memory for storing instructions and data at runtime. Each of these must be trusted, together with their interconnecting buses and the respective manufacturers.

This fueled research in architectures for on-single-chip trusted executions. For example, AEGIS [77] proposes a single-chip secure processor architecture that is robust against physical attacks. It uses Physically Unclonable Functions (PUFs) for reliable secret generation, and therefore does not require non-volatile memory to store secrets. AEGIS relies on a security kernel that handles memory management, multitasking and authentication mechanisms for secure code executions. However, the security kernel increases the TCB.

OASIS [78] delivers guarantees similar to AEGIS though with a reduced TCB. OASIS is a CPU instruction set extension that provides Trusted Computing services and limits the security perimeter to the CPU package. However, it requires hardware support.

TrustLite [79] is another hardware security architecture specific for low-cost embedded devices. TrustLite enables the set up of isolated execution environments for software modules and mutually authenticated channels between them. However, besides requiring hardware support, it allows a limited number of memory protection regions.

Recently, single-chip solutions have been developed and commercialized, by ARM with TrustZone [80], and by Intel with the Software Guard Extensions (SGX) [189]. TrustZone is mostly deployed and used on mobile devices, though the specification is about security extensions for the ARM processors. TrustZone provides hardware support for isolating and running code in a "normal world" and in a "secure world". Although the technology does not provide trusted storage by default, thereby severely limiting its capabilities (e.g., attestation), it can be enhanced with a firmware-TPM [81] to deliver TPM-like services. In this thesis we will not consider TrustZone.

Intel SGX provides code isolation, identification and attestation on the fast CPU by means of dedicated instructions. The CPU alone does not clearly have the resources to load, identify and execute a large program and process its data on the chip, so it needs off-chip memory. The CPU uses off-chip memory by encrypting and authenticating any code and data of the trusted execution that is stored to, or retrieved from, main memory. This allows SGX to maintain the trust boundaries within the CPU package, thereby considering off-chip memory as untrusted.

Finally, Sanctum [82] is a set of hardware extensions similar to SGX that has been proposed for a RISC-V CPU. Sanctum improves over SGX in at least two ways. First, it defends against some side-channel attacks, while SGX excludes side-channel attacks from the threat model. Second, its open implementation is *"easier to analyze than SGX's opaque microcode"* [82].

### 2.1.4 Today's Trusted Computing

Today[1], diverse hardware and architectures are available for trusted executions. We list those that we believe are the most popular.

First, Intel SGX is available on commercial CPU based on the Skylake and Kaby Lake microarchitectures. Also, SGX-based contributions are pullulating [38, 39, 40, 41, 49, 50, 63, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93].

AMD recently released processors based on the Zen microarchitecture, which includes AMD Secure Memory Encryption and Secure Encrypted Virtualization [199, 200]. The former allows it to maintain data in main memory in an encrypted form, thereby preventing snooping attacks on the bus and making the CPU package the security perimeter within which data is processed in cleartext. The latter provides support for encrypted virtual machines, thereby protecting their execution from physical attacks and also other software running on the platform. The objective

[1] at the time of writing this thesis, i.e., early 2019

is to secure virtual machines from a potentially malicious hypervisor, while the objective of SGX is to secure application-level code; this makes us believe that trusted executions with a small TCB on AMD technology would follow an approach similar to XMHF-TrustVisor. Also, unlike Intel's solution, AMD SME and SEV use an ARM-based secure processor integrated within the chip [199]. Software on the CPU communicates with the secure processor firmware using the SEV driver [200].

Meanwhile, TPMs have been upgraded from $v1.2$ to $v2.0$. The main difference with respect to $v1.2$ is the upgrade of cryptographic algorithms, for instance replacing SHA-1 with SHA-256, and the ability to implement new cryptographic algorithms. Also, it should be considered that TPMs have additional capabilities with respect to on-processor instruction set extensions, namely monotonic counters, useful for preventing rollback attacks, and NVRAM, access-controlled non-volatile memory useful to store (for instance) credentials. In addition, it is worthwhile noticing that Microsoft Windows 10 requires a TPM $v2.0$ for several services [190].

## 2.2 Background of Two Trusted Computing Architectures

In this section we provide some background on two classes of trusted executions by using two representative architectures: first, the *virtualization-based* class, which we describe using XMHF-TrustVisor [53, 94] (Section 2.2.1); second, the *instruction-based* class, which we outline using Intel SGX [189] (Section 2.2.2). The former class leverages hardware support for virtualization, particularly for virtualizing physical memory, to enforce isolation between the trusted and the untrusted execution environments and requires a hypervisor. The latter class instead leverages dedicated CPU instructions to create a secure environment, allocate memory for its exclusive use, add code and data, execute and attest the code. We believe that the virtualization-based class can also include the recent AMD Secure Encrypted Virtualization (SEV) technology, while the instruction-based class can also include the Sanctum hardware extensions [82].

The background serves to highlight architectural differences between XMHF-TrustVisor and SGX, such as memory management and entry/exit from a trusted execution, but also to show that they share common features at a higher level (such as isolated code execution and attestation). We will later abstract (Section 2.3) these features and use them as the foundation for the contributions of this thesis. In particular, our protocols in Chapter 3, (partially) in Chapter 4 and in Chapter 5 are described in terms of these primitives.

### 2.2.1 Background on XMHF-TrustVisor

XMHF-TrustVisor is a tiny hypervisor that provides efficient isolated execution and attestation of self-contained code. More precisely, TrustVisor is built as an "(hyper-)application" within the extensible modular hypervisor framework (XMHF). At boot time, XMHF uses the `GETSEC[SENTER]` instruction (on Intel processors) or `SKINIT` instruction (on AMD processors) to start the trusted hypervisor. These instructions are respectively part of Intel Trusted eXecution Technology (TXT) and AMD Secure Virtual Machine (SVM) and allow to launch a measured environment for a Virtual Machine Monitor (VMM)[2]. The measurement process involves computing a cryptographic hash (i.e., the identity) of the code before it is executed, and storing the identity securely to be later attested. The identity is secured by resetting a platform configuration register (PCR) on the TPM to 0—such a PCR reset denotes that the Dynamic Root of Trust mechanism was triggered to launch a measured environment at runtime—and extending it (i.e., $PCR \leftarrow hash(0||codeIdentity)$) with the computed code identity to build a hash chain. These PCRs can be later attested by the TPM, so that a client can check that the expected code (i.e., XMHF-TrustVisor's) was loaded and executed.

XMHF-TrustVisor uses extended/nested page tables (EPTs on Intel, or NPTs on AMD) to secure itself and any isolated code execution it performs. Such hardware page tables provide a further level of translation of the virtualized guest physical memory addresses into the actual physical memory addresses. Crucially, they also allow XMHF-TrustVisor to set up permissions, and thus memory access control mechanisms, for specific physical memory pages. In this way, the hypervisor can prevent the untrusted guest OS and other applications to tamper with the hypervisor's code/memory and with an isolated code execution. It clearly follows that the hypervisor must be trusted to behave properly.

XMHF-TrustVisor provides isolated code execution and attestation by leveraging nested page tables and hyper-calls as follows. The trusted execution environment can be created by registering the trusted application code through a registration hyper-call. Code memory pages are isolated from the untrusted OS using nested page tables to forbid access to the physical pages. Any access from untrusted code to isolated pages thus traps into the hypervisor. Only when the instruction pointer points to the registered code's entry point—by making a function call—the execution flow traps into the hypervisor that switches to secure mode to execute the registered code. The hypervisor takes care of marshaling I/O parameters in and out the trusted execution

[2] Intel TXT first launches an authenticated code module (ACM) signed by Intel. The ACM then measures and launches the VMM [75].

environment. The code executes until it terminates. Most importantly, it is never preempted and all input data is provided upfront. Termination occurs when the code attempts to execute code outside its isolated region. Hence, it traps into the hypervisor that switches to non-sensitive mode to run (and makes the output available to) the untrusted application code. In the untrusted environment, the (still) isolated code can then be unregistered through a hyper-call which makes the hypervisor remove the protections of the isolated pages and zero any sensitive data left in memory by the execution.

A client can establish trust in the remote hypervisor-based execution by verifying two attestations: one that vouches for the correct execution of XMHF-TrustVisor and for its public attestation key, and one that vouches for the correct execution of the registered code. The former is produced by the TPM—the hardware root of trust—to cover the PCR that contains XMHF-TrustVisor's identity and the public attestation key (possibly extended to the identity stored in the PCR). The latter is similarly produced by XMHF-TrustVisor, through its software micro-TPM ($\mu$TPM), to cover the identity of the registered code. The client can eventually verify that: (i) the expected registered code identity is signed by a private key that is linked to the expected identity of the hypervisor; (ii) both the identity and the key of the hypervisor are signed/attested by a TPM whose public attestation key is certified by a Certification Authority, which is known and trusted by the client. As a result, based on (i) and (ii), the client can make a trust decision, i.e., whether or not to trust the remote execution of the registered code and to accept the results. We refer the reader to [53, 94] for additional details.

### 2.2.2 Background on Intel SGX

Intel SGX is an instruction set extension available on the Intel Skylake and Kaby Lake microarchitectures [225, 226], which enable trusted code execution and verification. It uses an area in main memory, encrypted by the CPU, where code and data can be placed for secure processing. It does not require external chips for attestation. Hence, the CPU package delimits the physical security boundary.

The instruction set is extended by just two instructions: `ENCLU` and `ENCLS`. The instructions allow the execution of "user" and "system" functions respectively, so they work with different privilege levels. `ENCLU` executes user-level (non-privileged, i.e., ring-3) SGX leaf functions (e.g., `ENCLU[EENTER]`), while `ENCLS` executes privileged (ring-0) SGX leaf functions (e.g., `ENCLS[ECREATE]`). So a privileged OS driver can execute the privileged (`ENCLS`) functions but these do not allow the driver to tamper with a trusted execution. For brevity, we will refer directly to a leaf function without specifying the instruction and we will describe only the leaf functions that are most important for our contributions.

These instructions enable: memory management, transitions into and out of the trusted execution environment, dynamic memory allocations and attestations. The untrusted OS can allocate regions, called Enclaves, in protected main memory (called Enclave Page Cache, EPC) to run user-level code; the OS manages these regions by paging (encrypted) memory when necessary. Since enclaves are specifically designed to isolate and protect confidentiality and integrity of user-level code, this allows limiting the TCB to the CPU package and the user-level code that is protected, purposefully excluding the OS. The user-level application contains trusted code inside the enclave region and untrusted code outside such region. Execution transitions into and out of the enclave occur by means of user-level leaf functions, without OS intervention. Dynamic memory allocation requires cooperation between the OS and the enclave. In particular, system functions enable the OS to propose memory layout changes, while the enclave can validate and accept or deny such changes. Finally, an enclave can use SGX leaf functions to attest its identity locally (this aspect will be clarified later).

We now describe in more detail how SGX can be used to execute code securely. A secured area called Enclave can be created (`ECREATE, EADD, EEXTEND` leaf functions) to set up an execution environment for trusted application code. At enclave-creation time, a range of logical addresses (ELRANGE) can be specified for enclave use. Memory accesses within ELRANGE (must) translate to CPU-protected main memory pages; while accesses outside ELRANGE translate to untrusted memory pages, except for code access because the enclave cannot execute code that is outside its secure region [189, 2.5.2]. Hence, it is worth stressing that the enclave can access, but cannot execute instructions within the untrusted part of its application's memory. One or more Thread Control Structures (TCSs) can be included in the enclave. Each TCS contains one entry point, where the enclave can start executing, and one (or more) State Save Area frame (or SSA), where the architectural state of the enclave thread is stored on interruption. When the enclave is finalized (`EEINIT`), its identity—the `MRENCLAVE` register value—has been calculated and its code is ready to be executed.

The enclave runs by executing `EENTER` on a TCS. This enables enclave mode and transfers control to the entry point contained in the specified TCS. Entering (resp. interrupting) an enclave requires (resp. consumes) one SSA, and marks the TCS as busy (resp. available)—the processor has to know where to save the processor state securely if the enclave is interrupted; also, if the enclave is interrupted, another separate SSA must be available to re-enter it, again in case the new execution is interrupted. The enclave executes until it either terminates (EEXIT, synchronous exit) or is interrupted by an Asynchronous Exit (AEX). On AEX, the processor state (e.g., registers) is saved in a SSA frame inside the enclave, and replaced by a synthetic state to avoid leaking secrets when untrusted code resumes. The enclave can then be re-entered or resumed (ERESUME) through a TCS. If resumed, the processor state is restored from a SSA.

Adding and removing enclave memory pages at runtime requires cooperation between untrusted privileged code (i.e., an OS driver) and the enclave's trusted application code. The OS can add empty pages of protected main memory to the enclave (EAUG leaf function, which adds and zeroes a page); SGX then associates these pages to the enclave and marks them as pending. Since protected memory is encrypted, it cannot be accessed by untrusted code, and these pages can only be associated with one enclave instance, it follows that these pages are in fact isolated from any other code. For each page, the enclave can accept it (EACCEPT), or accept and copy into it the content of an available enclave page (EACCEPTCOPY); in both cases, SGX clears the pending bit; most importantly, the enclave can access the page only once it has accepted it. Similarly, the OS can reclaim enclave pages by changing their type (EMODT). The change must be accepted by the enclave first, then the OS can remove (EREMOVE) the pages.

SGX uses a remote attestation capability to prove to a client that an enclave has been executed in a trusted environment. The remote attestation is based on asymmetric cryptography and it is performed by a special Quoting Enclave using its private key, while the associated public key is kept by the Intel Attestation Service (IAS) [194]. The Quoting Enclave is meant to attest the identities of other enclaves running on the same platform. In particular, it does so by converting the local attestation by an enclave into a remote attestation. The local attestation allows SGX enclaves to prove to other enclaves that they (both the attested enclave and the verifier enclave) run on the same platform. The local attestation is based on symmetric cryptography and the secret key is secured by the CPU.

The two-step attestation procedure [194] thus works as follows. In the first step, the enclave produces a local attestation (using the EREPORT leaf function) as output. Such local attestation includes the enclave's identity and some small data (REPORTDATA, a $64$ bytes long data that is possibly the hash of the received input and the produced output data). Untrusted code then forwards this local attestation to a special Quoting Enclave. In the second step, the Quoting Enclave produces as output a remote attestation for the original enclave using an Enhanced Privacy Identifier [95] (EPID) signature. This remote attestation is eventually forwarded to the remote party and verified by contacting the IAS.

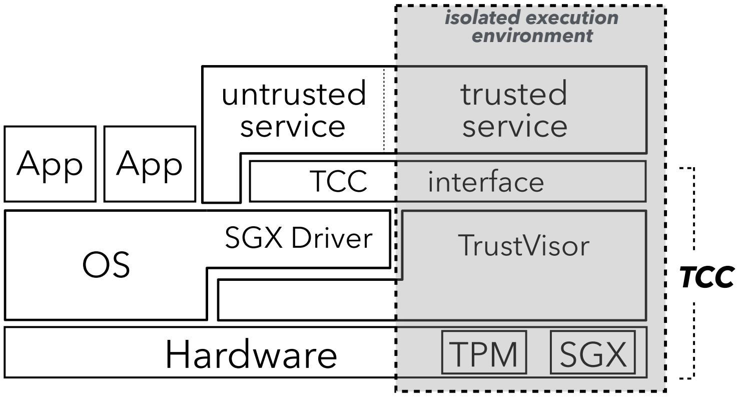


Figure 2.1: System design and trusted component interface.

We mention that EPID uses group signatures that allow the signing platform to stay anonymous. EPID [95] is an extension of the Direct Anonymous Attestation (DAA) [96]. DAA has been implemented on TPMs, for which (in their early versions) privacy issues due to attestations were already known. EPID improves on DAA by enabling key revocation without reducing anonymity. Privacy is however out of the scope of this thesis.

For further details on Intel SGX, we refer the reader to [189, 194].

## 2.3 An Abstraction of the Trusted Computing Component

As the Trusted Computing area is relatively new and fast changing (see Section 2.1), we believe that working on a specific technology has drawbacks. In particular, architecture-specific contributions would not be able to retrofit existing hardware, nor be a reference for future architectures. So these contributions end up depending on technology that may quickly become obsolete. Also, optimizations that work on an architecture may not apply, or be easily implemented, on others. Finally, these contributions would not provide insights on the fundamentals of trusted executions.

For example, XMHF-TrustVisor [53, 94], Flicker [52], SGX [189] and SEV [200] are different architectures that share a common feature, i.e., they all enable trustworthy and verifiable remote executions. However, it is not clear for instance how an application that is secured using SGX could instead leverage XMHF-TrustVisor, or a future version of the hypervisor based on AMD's secure virtualization technology, or the Sanctum extensions. As another example, the ability to allocate/load memory dynamically at runtime allows to avoid loading a large block of memory upfront; however, in XMHF-TrustVisor, the hypervisor can take care of dynamic allo-

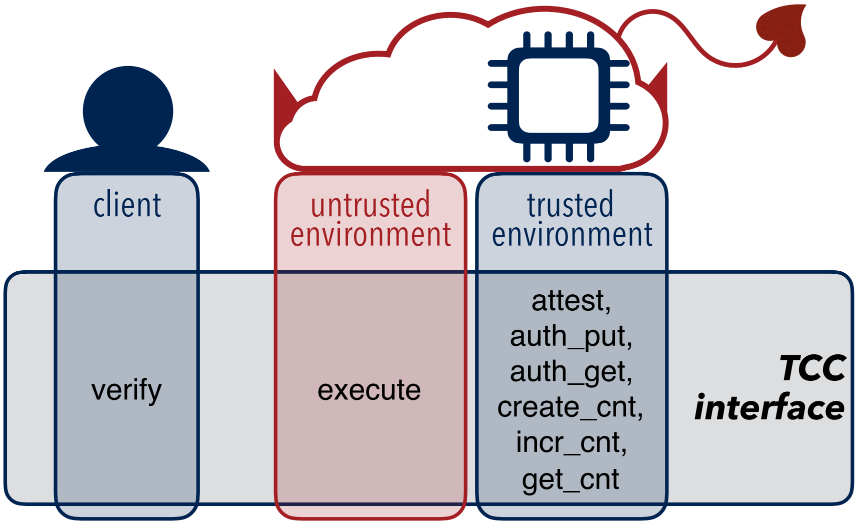


Figure 2.2: Which code calls which primitive.

cations/loading because it is trusted, while in SGX the procedure requires cooperation with the trusted application.

For these reasons we abstract the trusted computing component (TCC, Figure 2.1), both hardware and software, with a narrow interface. Such an interface consists of a small set of primitives (Section 2.3.1) that we derive directly from well-known concepts such as isolated execution, code attestation and attestation verification. Such an interface therefore allows us to fully hide the details of both trusted hardware and software support for trusted execution.

### 2.3.1 TCC interface

The primitives that constitute our TCC interface are depicted in Figure 2.2, also showing which code calls each primitive—and from what execution environment. The interface logically extends through different execution environments, namely: in the untrusted environment of the cloud provider, untrusted code is allowed to (call a primitive to) trigger an isolated execution; in the trusted environment of the cloud provider, code executing in isolation can make the TCC perform a code identity attestation, for example; finally, the remote client can perform an attestation verification to make a trust decision.

A natural question is whether such an interface is complete, i.e., whether the code executed either using XMHF-TrustVisor or Intel SGX would look the same, or additional primitives are necessary. The answer is that the interface is almost complete. Two additional mechanisms, for exiting the execution environment and accepting memory, are required and can be easily implemented. At the user-level, both require a simple "return" from a function in XMHF-TrustVisor,

| Chapter | | | Primitives | Parameters | |
|---|---|---|---|---|---|
| 3 | 4 | 5 | | input | output |
| ✓ | ✓ | ✓ | execute | code, input data | output data |
| ✓ | ✓ | ✓ | attest | input and output data identities, nonce | attestation |
| ✓ | ✓ | ✓ | verify[*] | remote attestation, code identity, input and output data identities, nonce, certified TCC public key | 0/1 (false or true) |
| ✓ | | | auth_put | recipient code identity, plain text data | encrypted data |
| ✓ | | | auth_get | caller code identity, encrypted data | plain text data |
| | | ✓ | create_cnt | service identifier | 0 |
| | | ✓ | get_cnt | | counter value |
| | | ✓ | incr_cnt | | counter value $+1$ |
| ✓[†] | ✓[†] | ✓ | get_cert | (none) | TCC's certificate |

[*] implemented at the client [†] not used explicitly, but assumed in the model

Table 2.1: Primitives used in this thesis. The columns indicate: the name of the primitive, the input and output parameters, the chapter (and so the technique) where a specific primitive is used.

though a special instruction in SGX. Hence, simple wrappers can solve the issue. The implementation of LaSt$^{\text{GT}}$ in Chapter 4 will elaborate more on this. We do not include these implementation details in the interface to simplify the description.

Our trusted execution primitives are detailed in Table 2.1. The primitives do not include the enhancements that are presented in later chapters of this thesis.

Beginning from the "Chapter" columns, we highlight that the primitives execute, attest and verify are common throughout the techniques in this thesis; auth_put and auth_get are exclusively used to enable our technique (Chapter 3) for executing large code bases; create_cnt, get_cnt and incr_cnt are specifically required for secure and available passively replicated executions (Chapter 5). We remark that get_cert is explicitly used in Chapter 5 while the other techniques implicitly use it by assuming its functionality in their respective system models.

Next, we briefly describe each primitive and then present their implementations.

• execute makes the TCC execute a code over some input data and eventually returns a result as output. Both the code and the input data are provided as input parameters. The TCC is responsible to identify the input code by hashing its binary and to store the identity securely in an internal register. This identity is then used for code attestation.

We point out that the code to be executed is self-contained. This is a natural requirement since the execution environment is strongly isolated and the code cannot rely on untrusted code, which is outside the trusted computing base (TCB), so neither identified by the TCC nor verified by the client. For example, although it would be tempting to enable system calls to the untrusted OS from the isolated environment, this would enlarge the attack surface with hundreds of system calls that are difficult to secure, making the system susceptible to Iago attacks [97] (i.e., system calls that return values crafted by a malicious kernel so to induce the running code to undertake an arbitrary computation). Hence, any service application executed on the TCC needs to be self-contained, i.e., with statically linked libraries and no OS dependencies.

• attest makes the TCC produce an attestation using the TCC's private key over the identity of the code (say $c$), which is loaded and identified through the execute primitive. The attestation allows binding the identities (or integrity measurements) of the input and output data together with the identity of the code $c$, which is running on the TCC. The input data includes a client-provided nonce and it is received by the code through the execute primitive, while the output data is the result of the execution of $c$. The executing code $c$ is responsible (and must be programmed) to compute the integrity measurements (i.e., hashes) of the input and output data before calling the attest primitive and supplying these measurements as parameters.

• verify is implemented at the client and accepts as input an attestation and the respective nonce, a certificate that vouches for the TCC's public attestation key, and the execution parameters such as the code identity and the measurements of the input and output data. In XMHF-TrustVisor, the TCC's public attestation key belongs to the hypervisor; the key and the hypervisor identity are verified using a TPM attestation. The verification is successful and returns true if the TCC's public key is certified by a trusted Certification Authority and the provided parameters are the intended ones. Otherwise, it returns false and the client can make the decision to reject the received results.

So far we have describe the basic primitives for trusted executions, that simply allow to execute, attest and verify some code. We remark that our contribution in Chapter 4 only uses these primitives, though an enhanced version of execute that we describe later. Next, we extend this basic set with primitives for secure storage and trusted counters. The primitives for secure storage, also enhanced later, enable our contribution in Chapter 3, while those for trusted counters enable our contribution in Chapter 5.

• auth_put and auth_get provide functionality for identity-dependent secure storage using a secret key stored inside the TCC. The former (auth_put) allows to protect some input data on the behalf of the currently running code. It requires the code running on the TCC to specify the identity of a recipient code that is allowed to retrieve the data when the recipient code will later execute on the TCC. Similarly, the latter (auth_get) allows to validate some protected input data. It requires the code running on the TCC to specify the identity of the previously executed code that originally put the data in secure storage. Notice that both primitives work with the identities of the sender and of the recipient code, so the data is bound to two identities. Also, the TCC directly computes the sender code identity in auth_put and the recipient code identity in auth_get, so the identity of the running code is always included by the TCC.

- create_cnt, incr_cnt and incr_cnt provide functionality for trusted counter management. In particular, the TCC creates, stores and modifies pairs of (counter identifier $cid$, value), where the identifier depends on the running code's identity (computed by the TCC) and defined as $cid \leftarrow h(code\ identity||sid)$. Here the $sid$ represents a service identifier. All the primitives thus accept a $sid$ as input and finally return the last or the incremented counter value.

- get_cert returns the public key (or the certificate) of the TCC. Such key enables a remote party to verify the attestations issued by the TCC. It should be signed such that the resulting trust anchor is a trusted root certification authority—possibly the manufacturer of the trusted hardware component.

### 2.3.2 Example Implementations of the Primitives

We now describe two implementations of the primitives using the XMHF-TrustVisor and Intel SGX architectures, which we introduced in Section 2.2. We use pseudo-code jointly with hypercalls (on XMHF-TrustVisor) or CPU instructions (on SGX) to describe their implementation. Such description style allows us to intentionally hide "pedantic" details, mainly related to the exact implementation of the I/O parameters of the primitives.

#### 2.3.2.1 Implementing execute

The execute primitive represents the front end for the entire trusted code execution. Algorithm 1 sketches an implementation for XMHF-TrustVisor while Algorithm 2 for Intel SGX. The I/O parameters are the same in both. This is expected since the primitive is supposed to set up a measured isolated address space with regions for various memory sections (i.e., text, data, stack), to let the code start executing at its entry point(s), to get its output (possibly containing an attached attestation) and return it. Let us elaborate now on some implementation-specific details.

**On XMHF-TrustVisor** (Algorithm 1) the main steps can be grouped into code registration and unregistration (lines 1, 5), I/O data marshaling and unmarshaling (lines 2, 4) and execution call (line 3). Only the first group involves interaction with the hypervisor for (un)registering (i.e., (un)isolating) code pages, respectively at the beginning and at the end of the implementation. The second group organizes the data (input data, nonce, heap memory for dynamic allocations, memory to contain output data, etc.) in a buffer that the hypervisor will take care of bringing in and out the trusted execution environment, respectively before the code execution starts and immediately after it terminates. The execution call (in the middle of the primitive) acts just like a function call, though it traps[3] into the hypervisor that switches control to the isolated code.

```
Input: description of code's text, data, stack
       sections, and input data (including nonce)
Output: output data
trigger registration hyper-call to isolate code
  and I/O memory pages
encode I/O parameters for I/O marshaling
call isolated code
decode output data
trigger unregistration hyper-call to return
  isolated memory pages
return output data
```

**Algorithm 1:** execute primitive on XMHF-TrustVisor. The implementation uses the original version of the hypervisor.

```
Input: description of code's text, data, stack
       sections, and input data (including nonce)
Output: output data
 1: init SGX Enclave Control Structure (SECS)
 2: initialize TCS(s)
 3: trigger system call to ECREATE enclave
 4: for each code page to be isolated do
 5:     trigger system call to EADD enclave page
 6:     trigger system call to EEXTEND the page
        content to the enclave
 7: end for
 8: trigger system call to EENIT the enclave
 9: run EENTER on the enclave's TCS
10: forward local attestation to Quoting Enclave
    for remote attestation
11: trigger system call to EREMOVE the enclave
12: return output data
```

**Algorithm 2:** execute primitive on Intel SGX. The implementation is typical for applications that do not make use of asynchronous exits (AEX's).

**On Intel SGX** (Algorithm 2), CPU instructions are used, rather than a hypervisor, to secure and execute a piece of code in a so called Enclave. As before, we can spot the code isolation and un-isolation (lines 1-8, 11) and execution (line 9) phases though I/O is performed differently. Code isolation is performed by creating an enclave (lines 1-3), adding memory pages to it (line 5, i.e., associating pages of protected memory to the enclave and copying non-enclave memory content into them), extending them (line 6, i.e., in Trusted Computing terms, measuring/hashing them, concatenate the result with the `MRENCLAVE` register value in the SGX Enclave Control Structure (SECS), where the identity of the code is calculated, then hash the concatenation and store the result into `MRENCLAVE`) and finally initializing the enclave (line 8). Un-isolation is performed by removing enclave memory (line 11, i.e., un-associating its pages from its SECS). All these operations require privileged instructions and must therefore be executed at the OS level, possibly

[3] A trap is not considered as an interaction with the hypervisor.

implemented by an ad-hoc driver. In our pseudo-code, we included a system call for each operation for clarity. However other implementations may opt to move some complexity onto the driver in order, for instance, to set up an enclave using a single system call. The code execution phase instead occurs through an application-level instruction (line 9), so it does not require an OS driver.

Data I/O is managed by using the capability of an enclave to access the untrusted part of the address space—thus not shown in the algorithm. In particular, non-enclave code can supply/receive data to/from the enclave by letting the enclave read/write data in untrusted memory. It is up to the enclave code to validate/protect the data.

Finally, the attestation step (line 10) converts a "local" attestation into a "remote" attestation—recall Section 2.2.2. We provide some details of the enclave execution for clarity and to better explain the difference with respect to the attest primitive in SGX. While the enclave is running, the enclave calls the attest primitive to get a local attestation—which proves to the Quoting Enclave that the locally attested enclave runs on the same platform. When the enclave terminates, it outputs the local attestation. At line 10, untrusted code forwards the local attestation to the Quoting Enclave for converting it. The Quoting Enclave outputs the remote attestation, which will be verified by the client. The remote attestation then replaces the local attestation in the output data of the execute primitive.

#### 2.3.2.2 Implementing attest

Two possible implementations are sketched in Algorithm 3 and Algorithm 4 taking the same input parameters. The attestation parameter (i.e., the second input parameter in the algorithms) represents an unambiguous succinct description of the I/O data that must be bound to the code identity. It can be defined as the hash of the input and the output data and must be computed (or available, e.g., when the input is loaded and validated on demand) at runtime in the trusted execution environment. Notice that the attestation completely describes the computation that is performed in the trusted environment. In fact, the identity of the code is included in the attestation by the TCC, while the attestation parameter and the nonce are added by the attest primitive. So at verification-time, if the verification is successful, this allows the client to know precisely what has been computed remotely.

A nonce can be included in the attestation to deliver freshness guarantees. Such nonce can be generated by the verifier (e.g., a client), forwarded to the remote untrusted platform and transferred to the trusted code inside the isolated execution environment as input to the execute primitive. This allows the verifier to prevent attestation replay attacks. However, in the case of replay-insensitive computations (e.g., the sum of all integers in a large dataset), using a default public nonce enables caching computation results and attestations that can be verified by several clients. So, clients can still get the same security guarantees but the untrusted provider can save computing power, trusted hardware resources and provide rapid responses.

**On XMHF-TrustVisor** the implementation (Algorithm 3) is heavily based on the attestation protocol in [53]. The hypervisor sets up, manages and exposes primitives for a software-based micro-TPM ($\mu$TPM). The $\mu$TPM provides a set of Platform Configuration Registers ($\mu PCR$'s), initially set to zero, where a $\mu PCR$ is the equivalent of a TPM's PCR. The $\mu$TPM modifies these registers by extending them with an integrity measurement $im$ received in input, for example, $\mu PCR[0] \leftarrow \mathsf{h}(\mu PCR[0]||im)$, thereby forming a hash chain. When a piece of code is registered (recall Section 2.2.1) in the hypervisor, the $\mu$TPM extends a $\mu PCR$, say $\mu PCR[0]$, with the identity of the code. The service code running in the trusted environment can extend other registers (say $\mu PCR[1]$, line 1) with the attestation parameter and then call the attestation hyper-call (line 2) supplying the nonce and the indexes of the $\mu PCR$'s to be attested for client verification. The attestation is finally returned (line 3) to the service code. The service code is ultimately responsible to return the attestation as output data.

It must be noted that this attestation alone is not sufficient, because the $\mu$TPM only provides a "software" root of trust, and XMHF-TrustVisor itself (and its public attestation key) must be attested by the hardware TPM and later verified. The TPM attestation is critical for the robustness of the trust chain. However, in order to prevent slowdowns due to the hardware TPM, it is separate from the code execution and thus not shown here. The hypervisor attestation can be performed by untrusted code specifying the relevant TPM's PCR registers that store the hypervisor's code identity and its attestation key measurement, and then the attestation can be forwarded to the client for verification.

**On Intel SGX** (Algorithm 4) the primitive is essentially a wrapper for the "local" attestation procedure that is executed using the `EREPORT` instruction. First of all, the instruction does not accept the nonce directly but just a 64-byte buffer (`EREPORTDATA` structure). So, the primitive hashes together (line 1) the input nonce and the attestation parameter, and supplies the result to the instruction (line 2). The local attestation has to be produced targeting (i.e., specifying the identity of the enclave that verifies the local attestation) the Quoting Enclave which later converts it into a remote attestation.

This means that the primitive only returns (line 3) a local attestation to the enclave. It is

```
Input: nonce, attestation parameter
Output: output attestation
trigger extend hyper-call to extend $\mu PCR[1]$ with the attestation parameter
trigger attestation hyper-call supplying the nonce to get the attestation of $\mu PCR[0], \mu PCR[1]$
return attestation
```

**Algorithm 3:** attest primitive on XMHF-TrustVisor. The identity of the service code is stored in $\mu PCR[0]$.

```
Input: nonce, attestation parameter
Output: output (local) attestation
$n \leftarrow$ hash(nonce || attestation parameter)
run EREPORT supplying $n$ and the Quoting Enclave's identity
return attestation (i.e., the report structure)
```

**Algorithm 4:** attest primitive on Intel SGX. The identity of the service code is stored in the $MRENCLAVE$ register inside the enclave, and then included in the report structure.

expected that the enclave later outputs such a local attestation on termination (i.e., in the execute primitive, Algorithm 2) and that untrusted code then forwards it to the Quoting Enclave so to be verified and converted—this is implemented in the execute primitive in Algorithm 2. The output of the Quoting Enclave (i.e., the remote attestation) can then be forwarded to the client for verification.

#### 2.3.2.3 Implementing verify

The primitive is meant to be performed by a trusted verifier, which is represented by the client in our model. Consequently, there is no trusted hardware (TPM, SGX, etc.) involved in the verification procedure. We sketch two examples implementations.

**On XMHF-TrustVisor** as detailed in [53], the primitive (Algorithm 5) performs one verification (line 1) for the TPM attestation—of the hypervisor code—and another for the hypervisor attestation—of the executed code. We include the former in the primitive for clarity, although it can be performed before the execution to verify the hypervisor's identity and its public key. The latter can be optimized by attesting and verifying a symmetric secret key in order to establish a secure channel between the client and the trusted code execution for data authentication.

**On Intel SGX** the primitive (Algorithm 6) checks the fields of the attestation from the Quoting Enclave (line 1) and, if successful, also contacts the Intel Attestation Service (IAS) [194] to verify the digital signature performed with the Enhanced Privacy ID (EPID) scheme (line 4). If both verifications succeed, then the remote enclave execution and its I/O data are the intended ones (line 9) and the client can make a positive trust decision. Otherwise (line 2 or 7) the attestation is not correct and the client should consider any received data as untrustworthy and reject it.

For brevity and clarity, we intentionally skip two steps: (i) the private EPID key provisioning protocol, to provide a private attestation key, and (ii) the registration at the Intel Attestation

```
Input: nonce, output hash, input hash, code identity, hypervisor identity, hypervisor public attestation key, TPM attestation, certified TPM public attestation key, hypervisor attestation
Output: true or false
if TPM attestation can be validated using certified TPM public attestation key $\wedge$ attested hypervisor identity is expected $\wedge$ attested hypervisor public attestation key is expected $\wedge$ hypervisor attestation can be validated using hypervisor public attestation key $\wedge$ nonce, code identity, hash(input hash$||$output hash) are expected then
    return true
else
    return false
end if
```

**Algorithm 5:** verify primitive for XMHF-TrustVisor. The implementation uses the original version of the hypervisor.

```
Input: nonce, output hash, input hash, code identity, remote attestation (from Quoting Enclave), public report key
Output: true or false
if code identity (i.e., MRENCLAVE register value inside the remote attestation) not expected $\vee$ hash(nonce$||$input hash$||$output hash) (i.e., REPORTDATA structure inside the remote attestation) not expected then
    return false
end if
contact IAS [220] through APIs [221]
validate attestation report from IAS with public report key
if IAS returns attestation not valid then
    return false
end if
return true
```

**Algorithm 6:** verify primitive for Intel SGX. The implementation is typical for SGX applications.

Note. Inside an enclave the verification is slightly different as follows: the signed Attestation Verification Report (returned by the IAS) is forwarded to the enclave, who performs the same checks as above and additionally verifies the signature, so it requires one additional input i.e., the public Report Key. See the implementation of the get_cert primitive on SGX for additional details.

Service (IAS) to verify the attestations. (i) is necessary to provide the Quoting Enclave with a private key which is used to convert local attestations into remote attestations by signing them. (ii) is necessary since it is required by Intel and *"only the IAS can verify the signature"* [219]. We believe these one-time procedures are mainly related with how Intel intends SGX to be used and with Intel's business relationships with its customers. As such topics are out of the scope of this thesis, we refer the reader to [194, 219] for more details.

#### 2.3.2.4 Implementing auth_put

The primitive is called by the code running in the isolated environment to protect input data that can be later authenticated by a recipient code (i.e., code that will be later run on the TCC using the execute primitive and that will receive the protected data), whose identity is also provided as input to the primitive. This primitive will be useful, and is also enhanced, in our contribution in Chapter 3. Two implementations are sketched in Algorithm 7 and Algorithm 8.

**On XMHF-TrustVisor** , the primitive (Algorithm 7) is implemented by means of the sealing operation performed by the software-TPM inside the hypervisor. The running code seals (line 1) the data, and the operation ensures that only the recipient code is able to unseal the data when it runs. The software-TPM creates an encrypted data blob containing the data and information about the sender and the recipient code identities (resp. $digestAtCreation$ and $digestAtRelease$ fields inside the blob). The blob is then returned to the running code (line 2) so to be later transferred to the intended recipient code. The hypervisor-managed software-TPM will later enforce access control to plain data at unseal-time.

```
Input: (recipient) code identity, data
Output: secured data
 1: trigger hyper-call for sealing data specifying
    the recipient identity[a]
 2: return secured data
```

**Algorithm 7:** auth_put primitive for XMHF-TrustVisor. The implementation uses the original version of the hypervisor.

```
Input: (recipient) code identity, data
Output: secured data
    (following [98])
 1: if there already exists a sealed
    public/private Diffie-Hellman key pair
    then
 2:     retrieve it from shared memory and
        unseal it using EGETKEY
 3: else
 4:     generate a public/private Diffie-Hellman
        key pair
 5:     seal it using EGETKEY and save it in
        shared memory
 6: end if
 7: run EREPORT to produce a local attestation
    of the public key for the recipient code
    identity
 8: send attestation through shared memory
 9: retrieve the recipient's local attestation of its
    key, and its key, from shared memory
10: run EGETKEY to retrieve the report key
11: verify that the local attestation belongs to
    the intended recipient and that it attests the
    received key
12: compute the secret key shared with the
    recipient code
13: use key to authenticate/encrypt data
14: return secured data
```

**Algorithm 8:** auth_put primitive for Intel SGX. The implementation follows the guideline in [98].

**On Intel SGX** the primitive (Algorithm 8) is more complex since the EGETKEY instruction only provides a secret sealing key to the running enclave, so it does not allow securing data to be transmitted to another enclave. We use the technique in [98] to run a protocol for exchanging public Diffie-Helmann keys between enclaves, so that they can create a secure channel between

[a] by suitably setting a $digestAtRelease$ value that allows unsealing the data only when $\mu PCR[0]$ (i.e., the identity of the running code) will match the recipient code identity.

them. If a key pair was previously sealed by the enclave, that pair is retrieved and used (lines 1-2). Otherwise, the protocol lets an enclave generate a public/private key pair and put it in sealed storage (lines 3-5). The enclave then generates a local attestation of its public key for the recipient code (line 7), so that they can establish a secure channel. Message passing between enclaves and persistent storage of sealed data are performed by making the enclave write/read protected data in untrusted shared memory (lines 8-9), where it is then managed (or transferred elsewhere) by untrusted code. After retrieving the local attestation of the recipient code and its public key, the enclave verifies the local attestation (lines 10-11) to establish trust in the recipient code. If successful, the enclave possesses the key of the intended recipient code and can therefore compute a shared secret for protecting the data on the channel (lines 12-13).

*A note on the initialization of a shared key.* It may appear that there exists an initialization issue in the first call of auth_put and auth_get in SGX. auth_put needs to access a shared key for protecting the data, and the protected data is then returned as output and sent on the channel. So the code in auth_get should first help generating such key and then use it on the received data. This means that the first call to auth_get cannot have the protected data immediately available as input. This is not an issue but rather an implementation detail that can be addressed in multiple ways. We mention two of them. (1) The data parameter could be a pointer to the data. In this case, auth_get only receives a pointer as input, and then accesses the data once the key is available. (2) The data parameter could include a type field and a payload field. The INIT type could be used in the first call without payload, to allow auth_get to retrieve the shared key, while the PROTECTED type with the protected input data in the payload field could be used in subsequent calls.

#### 2.3.2.5 **Implementing** auth_get

The primitive mirrors the behavior, and so the implementation, of the auth_put primitive. Even this primitives will be used and enhanced in our contribution in Chapter 3. For brevity, we only highlight the noteworthy parts of the implementations.

**On XMHF-TrustVisor** the primitive (Algorithm 9) performs a second level of access control—the first is performed by the software TPM inside the hypervisor to check if data unsealing (line 1) is authorized (i.e., if the running code's identity matches the identity of the intended recipient code). The primitive simply checks that the unsealed data was indeed sent by the intended code (lines 2-3). Only in this case it returns the unsealed data to the calling code (lines 4-5), otherwise the unsealed data is discarded as it is considered untrustworthy (line 8).

**On Intel SGX** the primitive (Algorithm 10) instead exactly mirrors the auth_put primitive, so we

```
Input: (sender) code identity, secured data
Output: validated data or $\perp$
trigger hyper-call for unsealing data [a]
if data has been unsealed then
    use sender code identity to compute $expected\_digestAtCreation$
    if $digestAtCreation$ matches $expected\_digestAtCreation$ then
        return data
    end if
end if
return $\perp$
```

**Algorithm 9:** auth_get primitive for XMHF-TrustVisor. The implementation uses the original version of the hypervisor.

[a] The hyper-call decrypts the data blob and checks that the currently executing code is authorized to access it—in our case by using $\mu PCR[0]$ to check the $digestAtRelease$ in the data. If (and only if) so, it returns the data and a $digestAtCreation$ value that the running code can use to check the identity of the code that originally sealed the data.

```
Input: (sender) code identity, secured data
Output: validated data or $\perp$
    (following [98])
if there already exists a sealed public/private Diffie-Hellman key pair then
    retrieve it from shared memory and unseal it using EGETKEY
else
    generate a public/private Diffie-Hellman key pair
    seal it using EGETKEY and save it in shared memory
end if
run EREPORT to produce local attestation of the public key for the sender code identity
send attestation through shared memory
retrieve the sender's local attestation of its key, and its key, from shared memory
run EGETKEY to retrieve the report key
verify that the local attestation belongs to the intended sender and that it attests the received key
compute the secret key shared with the sender code
use key to validate/decrypt data
if data is valid then
    return data
else
    return $\perp$
end if
```

**Algorithm 10:** auth_get primitive for Intel SGX. The implementation follows the guideline in [98].

avoid further details. We refer to the auth_put primitive (Section 2.3.2.4) for a discussion on the initialization of the secret shared key used inside the primitive.

#### 2.3.2.6 Implementing create_cnt

The primitive allows to create a trusted monotonic counter at runtime. The primitive is used in our contribution in Chapter 5. Next, we sketch two implementations. Another implementation (for this and the other primitives for trusted counters) can be put in place by adapting a more recent technique from Ariadne [99].

**On XMHF-TrustVisor** the primitive (Algorithm 11) is just a wrapper of a new hyper-call that we implemented for creating a trusted counter (line 1) within the hypervisor. For completeness, we provide some details of the hyper-call internals. The hyper-call implementation stores and modifies pairs of $(counterID, value)$, where the counter identifier depends on the running code's identity (i.e., the value stored in the platform configuration register $\mu PCR[0]$ inside the software micro-TPM of the hypervisor). More precisely, $counterID \leftarrow hash(\mu PCR[0]||uID)$, where $uID$ is a user-defined identifier. So only well-identified code can have read/increment access to the counter.

**On Intel SGX** it does not appear (from the specifications [189]) that trusted counters are available on the CPU. In the software development kit (SDK) [191] however, Intel provides support through a Platform Specific Enclave (PSE) for such counters, which are apparently stored on the Intel Management Engine.

The primitive (Algorithm 12) leverages such support to create a counter (line 1). We specify a restrictive owner policy that allows enclaves with the same measurement (i.e., with the same identity, and so built with the same code) to access the counter. Also, we seal (lines 5-6) the user-defined counter ID with the counter ID received from the counter creation call—a $sgx\_mc\_uuid\_t$ structure with a 3-bytes ID and a 13-bytes nonce[4]—and we save the sealed data (line 7). The reason for this procedure is that (i) the enclave needs the counter ID to access the counter, so this has to be saved, possibly in untrusted memory—this is secure since access to the counter is controlled by also checking the enclave's identity—and (ii) the application in the enclave can request more counters through the user-defined counter ID. So with this procedure, we ensure such user-defined ID is paired with the counter ID, which is not returned as output to the running code.

#### 2.3.2.7 Implementing get_cnt

The primitive allows to retrieve the current value of a trusted counter at runtime. This is used in our contribution in Chapter 5. Two implementations are sketched next.

**On XMHF-TrustVisor** , the primitive (Algorithm 13) is a wrapper of a new hyper-call (line 1) that we implemented for retrieving a trusted counter value within the hypervisor. In the following we detail the internals of the hyper-call. As with the hyper-call for creating a counter, this implementation computes the counter identifier using the running code's identity and the received user-defined counter identifier (see Section 2.3.2.6). So only well-identified code can retrieve the counter's value. The code in the hypervisor checks whether such counter has been previously created by looking up the pair $(counterID, value)$. If so, it returns the value, or an error (i.e., a default value) otherwise. After the hypervisor returns a value, the primitive behaves similarly

[4] Definition of nonce from SDK: *"An arbitrary number used only once to sign a cryptographic communication."*

```
Input: user-defined counter ID
Output: 0 or error
1: trigger hyper-call for creating the counter
   supplying the user-defined counter ID
2: if error creating counter then
3:     return error
4: end if
5: return result (i.e., initial value 0)
```

**Algorithm 11:** create_cnt primitive for XMHF-TrustVisor.

```
Input: user-defined counter ID
Output: 0 or error
1: call $sgx\_create\_monotonic\_counter\_ex$[a]
   with owner policy $0x2$[b]
2: if error creating counter then
3:     return error
4: end if
5: run EGETKEY to get seal key
6: seal user-defined counter ID, the monotonic
   counter ID (from previous call), the counter
   value
7: save sealed blob in untrusted memory
8: return value (i.e., initial value 0)
```

**Algorithm 12:** create_cnt primitive for Intel SGX.

[a] $sgx\_create\_monotonic\_counter$ only specifies a default policy.

[b] So that *enclave with same measurement can access the monotonic counter* [191].

when it continues at line 2.

**On Intel SGX** the get_cnt primitive (Algorithm 14) leverages the SDK support, following the same steps of the counter creation primitive (Section 2.3.2.6). First, if the counter ID is not available, it attempts to read a sealed blob from untrusted memory, trying to unseal it and to match the user-defined ID with the received one to retrieve the counter ID (lines 1-7). It then reads the counter (line 9) using the counter ID (in the $sgx\_mc\_uuid\_t$ structure). If the call fails or the value does not match the sealed one, an error is reported (lines 10-11). Otherwise, the value is returned to the running code (line 13).

#### 2.3.2.8 Implementing incr_cnt

The primitive allows to increment a trusted counter at runtime. Like the previous primitives for trusted counters, this is also used in our contribution in Chapter 5. We describe two implementations.

**On XMHF-TrustVisor** , the primitive (Algorithm 15) wraps a new hyper-call (line 1) that we implemented for incrementing a trusted counter within the hypervisor. The internals of the hyper-call are implemented as follows. As with the create_cnt hyper-call (Section 2.3.2.6), the running code's identity and the received user-defined counter identifier are used to compute the counter identifier. So only well-identified code can retrieve the counter's value. The code in the hypervisor checks whether such counter has been previously created by looking up the

```
Input: user-defined counter ID
Output: counter value or error
trigger hyper-call for getting the counter's value supplying the user-defined counter ID
if error getting counter's value then
    return error
end if
return counter's value
```

**Algorithm 13:** get_cnt primitive for XMHF-TrustVisor.

```
Input: user-defined counter ID
Output: counter value or error
if the triple (user-defined counter ID, counter ID, value) is unavailable then
    read unsealed blob from untrusted memory
    run EGETKEY to get seal key
    try unsealing the blob, or report error
end if
if user-defined counter ID mismatch then
    return error
end if
call $sgx\_read\_monotonic\_counter$ supplying counter ID
if error reading or value mismatch then
    return error
end if
return counter's value
```

**Algorithm 14:** get_cnt primitive for Intel SGX.

pair $(counterID, value)$. If so, it increments the value, updates the pair and returns the value, or an error (i.e., a default value) otherwise. Similarly, after the hypervisor provides the value, the primitive either returns the incremented value or an error.

**On Intel SGX** the incr_cnt primitive (Algorithm 16) leverages the SDK support, following the same steps of the counter creation primitive (Section 2.3.2.6). First, if the counter ID is not available, it attempts to read a sealed blob from untrusted memory, trying to unseal it and to match the user-defined ID with the received one to retrieve the counter ID (lines 1-7). It then increments the counter (line 9) using the counter ID (in the $sgx\_mc\_uuid\_t$ structure). If the call does not succeed or the value does not match the sealed one (incremented by one), then it reports an error (lines 10-11). Otherwise it seals the incremented value in a new triple, saves it in untrusted memory (lines 13-15) and returns the incremented value (line 16).

[99] identifies a problem in these steps: the impossibility of reading or incrementing the counter due to a power-loss after the counter has been incremented and before the sealed blob is saved. This is not an issue here since the primitive is only built for and used by our V-PR system, for verifiable passive replication, presented in Chapter 5. In V-PR, the power-loss (i.e., crash) of a replica is already accounted for in the model and simply results in the inability to produce messages that can be validated. Such misbehavior is simply ignored by the available replicas.

```
Input: user-defined counter ID
Output: counter value or error
trigger hyper-call for incrementing the counter's value supplying the user-defined counter ID
if error incrementing counter's value then
    return error
end if
return incremented counter's value
```

**Algorithm 15:** incr_cnt primitive for XMHF-TrustVisor.

```
Input: user-defined counter ID
Output: counter value or error
if the triple (user-defined counter ID, counter ID, value) is unavailable then
    read unsealed blob from untrusted memory
    run EGETKEY to get seal key
    try unsealing the blob, or report error
end if
if user-defined counter ID mismatch then
    return error
end if
call $sgx\_increment\_monotonic\_counter$ supplying counter ID
if error incrementing or value $\neq$ previous value +1 then
    return error
end if
run EGETKEY to get seal key
seal user-defined counter ID, the monotonic counter ID (from previous call), the incremented counter value
save sealed blob in untrusted memory
return incremented counter's value
```

**Algorithm 16:** incr_cnt primitive for Intel SGX.

#### 2.3.2.9 Implementing get_cert

The primitive returns a key that allows to verify a remote attestation.

**On XMHF-TrustVisor** the primitive (Algorithm 17) is meant to return the public attestation key of the hypervisor (line 1), whose private counterpart is used by the hypervisor to issue remote attestations. We recall that, in order to ensure the robustness of the trust chain, the verifying party additionally needs the certificate of (or a certified) public Attestation Identity Key (AIK) of the hardware TPM, which allows to verify the TPM attestation of XMHF-TrustVisor's code identity and public attestation key.

**On Intel SGX** implementing the primitive is more complex due to the nonexistence of such a certificate or key. The verification of a remote attestation is in fact supposed to be performed through a remote attestation verification service called Intel Attestation Service (IAS [219, 221]). However, by analyzing the details, the IAS signs attestation reports with a private Report Key, whose associated public key can be downloaded by the ISV (Independent Software Vendor) who previously registered with Intel. The public Report Key is therefore our target key. So we let the primitive return it (Algorithm 18, line 1). We stress that the key does not allow to verify directly

```
Input: ∅
Output: public attestation key
  1: trigger hyper-call for retrieving the
     hypervisor's public attestation key
  2: return key
```

**Algorithm 17:** get_cert primitive for XMHF-TrustVisor.

```
Input: ∅
Output: public report key
  1: return public report key
```

**Algorithm 18:** get_cert primitive for Intel SGX.

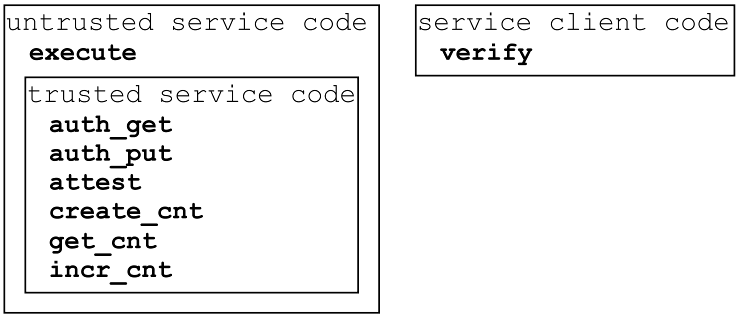


Figure 2.3: Primitives as they are used in the implementations of a service and service client.

the attestation, but it allows to verify the attestation report signed by Intel (i.e., a trusted entity), which states whether the verification was successful or not.

### 2.3.3 Primitives in Practice

**How the primitives are used.** We explain now how the primitive are used while programming a service for a trusted execution (i.e., where they are in the service code), and how they are used at runtime (i.e., the environment they are called from during the execution). Figure 2.3 shows a simplified diagram of the service and service client executables that only includes the primitives. The untrusted service code contains the trusted service code, that is transferred and executed inside the trusted environment, while the service client is implemented separately. As we mentioned, only the verify primitive is implemented inside the service client code. Also, only the execute primitive is called from the untrusted environment. The auth_get, auth_put, attest, create_cnt, get_cnt and incr_cnt primitives require the identity of the running code computed by the TCC, so they are called inside the trusted execution environment while the trusted service code is running.

We deliberately avoid to include the get_cert primitive. Although relevant, its implementation and execution could be separate from the service code and the service client code. In fact, the same key pair for attestation and verification can be used in different trusted executions.

**How our contributions leverage the primitives.** In Chapter 3, we replace a large monolithic code execution (using the execute primitive) with multiple executions of smaller code modules. Any two code modules, that are adjacent with respect to the intended execution flow, exchange data securely through the auth_put and auth_get primitives respectively, thereby forming a secure execution chain.

In Chapter 4, we design the execute primitive so that it can seamlessly handle large-scale data in main memory exploiting paged virtual memory. In particular, page fault interruptions are handled in the implementation of the primitive, and so "below" the TCC interface.

In Chapter 5, we use the primitives to make Passive Replication secure. The execute primitive is used to execute the service application at the primary replica and the state update service at the backup replicas. Both services are part of the same application (i.e., they are in the same binary), so the code executed at each replica has the same identity. The get_cert primitive is used to retrieve each replica's TCC certificate. The replicas use these certificates to verify each other's attestations. Then the create_cnt, get_cnt and incr_cnt primitives are used by the replicas to order the state updates and to agree on what replica is the primary.

# Chapter 3

# The Multi-Identity Approach for Identification of (only) Actively Executed Code

As we increase the size of the code and the data that we supply as input to the execute primitive (Section 2.3.2.1), security and efficiency issues arise. We will deal with large-scale data in Chapter 4. Now we focus on the issues raised by a large code base, which derive from how the identification of the code is performed in the primitive.

Code identification [22] is a key mechanism for guaranteeing execution integrity in Trusted Computing (Section 1.3.4). It consists of attesting the identity of some code $c$ within the Trusted Computing Component (TCC, Section 2.3) of an untrusted cloud provider's platform and then verifying both the attestation and the code identity on the client side. More precisely, the TCC computes $c$'s identity by hashing it, and attests the identity by digitally signing it. The attestation is then sent to the client who verifies that the intended code was executed. By extending the same procedure to the input and output data, the client is also able to verify that the received output was obtained by running $c$ with the intended input.

The major challenge with using code identification for securing increasingly complex software is that the overhead of computing $c$'s identity before execution grows linearly with $c$'s size. In particular, the overhead scales with the size of the code that *may* be executed, not the size of the code modules that *are* actually executed. Consequently, this becomes a concern when the actively executed code is only a fraction of the whole code base.

Previous work [38] circumvented this issue using an additional loader component but this has some drawbacks. The component is loaded and identified first, and then it later loads and validates on-demand other code to be executed. However, this enlarges the TCB due to the additional component. Also, the integrity of the component is only guaranteed at load time, when its identity is computed and stored by the TCC. The same argument holds similarly for the code that is loaded and validated later. Hence, although such one-time code identification has low overhead, it negatively impacts the freshness of the execution integrity guarantee.

In this chapter we present a protocol for code identification and execution that breaks the coupling between code size and cost of identification, and solves the trade-off between security and performance [67]. The protocol has two key desirable features. First, it allows the trusted architecture to load, identify and run only modules of the code base that are actually executed. Each module is identified as many times as it appears in an execution sequence. This provides execution flexibility to the cloud provider, saves TCC resources and strengthens the execution integrity property. Second, the correct execution sequence of code modules is ensured by a robust execution chain. Each module secures the application data using a secret key that depends on its own identity and the identity of the next module in the execution sequence. These two mechanisms combined enable a secure and efficient identification of the actively executed code.

The protocol further enables efficient verification on the client side. In fact, by only verifying the execution of a chain end-point, the client can establish trust in the entire execution chain. Noticeably and desirably, the client does not need to be aware a priori of the exact execution order. Also, unused code modules have to be neither loaded nor verified.

In addition, our protocol is agnostic to the details of the TCC, which makes our contribution generally applicable. The protocol is in fact based on a subset of the TCC primitives that we introduced in Section 2.3.1. More precisely, five primitives (including one on the client for execution verification) represent the bridge between the protocol and the Trusted Computing services provided by the TCC, such as isolated code execution, attestation and secure storage.

### Contributions

- We present an analysis of today's trend in trusted executions, how it is going to impact on their performance and what the current solutions are.
- We design an efficient protocol for loading and identifying (and then executing) only the code that is necessary to serve a client request. The protocol is built using a set of primitives that abstract a generic trusted component. The cost of code identification scales with the size of the modules that are executed, rather than with the size of the service code base.

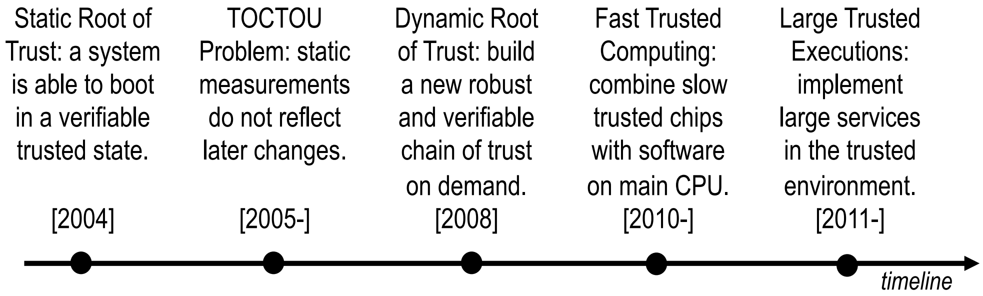

Static Root of Trust: a system is able to boot in a verifiable trusted state.
[2004]
TOCTOU Problem: static measurements do not reflect later changes.
[2005-]
Dynamic Root of Trust: build a new robust and verifiable chain of trust on demand.
[2008]
Fast Trusted Computing: combine slow trusted chips with software on main CPU.
[2010-]
Large Trusted Executions: implement large services in the trusted environment.
[2011-]
timeline

Figure 3.1: Trends in Trusted Computing research work show an initial focus on reducing the Trusted Computing Base (TCB), while recent advances in technology enable to secure entire unmodified services, thus enlarging the TCB.

- We analyze the security of our constructions. Also, we introduce a novel zero-round key sharing technique for code executions based on a trusted component. The technique allows two pieces of code, each one executed in isolation, to share a unique secret key using their identities. As the two executions do not need to exchange any message, this improves performance over current solutions. In addition, small changes are required to support the technique on current trusted components.
- We implement the protocol on a hypervisor-based TCC, namely XMHF-TrustVisor [53, 94]. We apply it to the widely-deployed SQLite DB engine [188], model the implementation for automatic verification using Scyther [100, 101], and show its performance benefits.

## 3.1 Towards Trusted Executions of Actively Executed Code

Current trends (Figure 3.1) in Trusted Computing evidence that the code used in trusted execution is growing. We show that this raises either efficiency or security concerns in Trusted Computing architectures, and that this is a result of how code identification[1] is done today [22]. This helps us define the problem statement, goals, and outline our solution.

### 3.1.1 Previous Work

Early work used trusted hardware to verify the integrity of a system's initial state [102, 103]. The mechanism involves identifying—taking integrity measurements, i.e., hashing—the software components (e.g., BIOS, boot loader, OS, applications) that bring the system into an operative state. The identities are stored on trusted hardware (e.g., a TPM) and conveyed to a client

[1] How an identity is assigned to the code to be executed. This identity can be later attested and verified remotely.

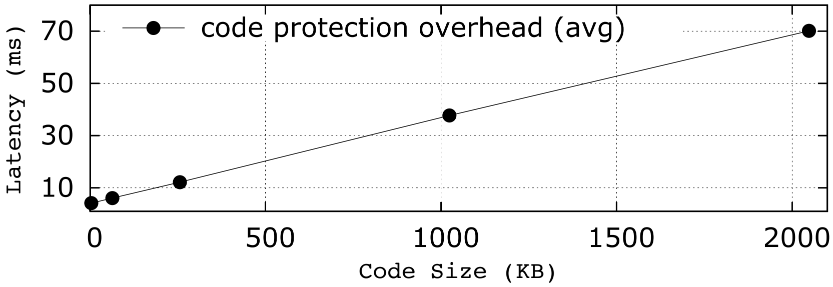


Figure 3.2: Security-sensitive code registration latency in XMHF-TrustVisor. It shows a linear dependence between code size and protection overhead.

through an attestation. The client bootstraps trust in the system's initial state by verifying the validity of the attestation and by matching the identities with the expected ones.

Preserving trust during the execution is hard. Operating systems are constantly subject to attacks; vulnerabilities are discovered on a daily basis [104]; and tools are available to exploit them [105]. Hence, the guarantee that a system is trusted at a certain point in time may not hold later—this is also known as time-of-check-time-of-use (TOCTOU) gap [73, 106].

This gap was reduced through the notion of *late launch* [76] to create a Measured Launch Environment on demand. Flicker [52] shows that the technology can be used to run a security-sensitive piece of code in isolation. The result is a dramatic reduction in TCB size and, consequently, of the attack surface. Since then, faster architectures have been devised [53, 54, 55, 78, 107].

Improvements in Trusted Computing technology have made it possible to grow the code base from a few kilobytes to hundreds of megabytes. In fact, in order to provide security guarantees to a broader set of applications, some projects have secured entire database engines [36, 37], and even unmodified Windows binaries such as SQL Server and Apache HTTP Server [38]. It thus follows that the complexity of software running in a trusted environment is growing.

### 3.1.2 Security or Efficiency, But Not Both

There are currently two alternatives to deal with such large code bases, and both come with downsides. We dub the first as *measure-once-execute-forever* [38]. The integrity measurement is taken only before the execution of the code, which then continues to run in the trusted environment indefinitely. Unfortunately, this approach brings us back to the TOCTOU problem. Since the integrity measurement of a code base is only taken once, it will not detect any successful attack that later compromises the application.

The second alternative is instead dubbed *measure-once-execute-once*. The measurements are repeated before each execution (e.g., an application based on Flicker [52]). This approach instead may raise efficiency issues. In fact, in order to assign an identity to the code, it must be loaded first and then hashed.

As an example, in Figure 3.2 we quantify this load-and-hash cost on XMHF-TrustVisor [53, 94]. We measured the time to register different code bases—the details of the experimental setting can be found in Section 3.4.1. During this procedure, the memory pages of the code are isolated and identified. The time scales linearly with the code size reaching about $37ms$ for just 1MB of code.

Such a linear dependence also holds for Intel SGX [107, 108], used to build secure Enclaves. In fact, recalling Section 2.2.2, after an Enclave is created (`ENCLS[ECREATE]` instruction), code pages must be added and measured (`ENCLS[EADD]`, `ENCLS[EEXTEND]`). Hence, the overhead of creating an Enclave identity grows linearly with the code size.

### 3.1.3 Problem Definition

Clearly code identification cost has become a bottleneck. On one hand, if the code is identified only *once*, identity integrity stales over time. One the other hand, if the code is identified *repeatedly*, the overall execution time may increase considerably for large code bases. The ideal balance is to have non-stale identities and an execution time less dependent from code base size.

In this thesis, we aim at making the secure execution cost scale with the size of the code modules as they are executed, rather than the size of the code base as a whole, independently from the trusted component in use. Such generally-applicable method can reduce the cost of re-identifying some code, refresh the integrity guarantees, and also reduce the size of the active TCB.

In summary, we seek to attain the following properties:

1. *Secure proof of execution*. The proof of execution of the correct code must be unforgeable, unambiguous, and linked to the hardware root of trust.
2. *Low TCC resource usage*. The protocol should achieve security with minimal resource (code identification, cryptography, storage, etc.) demand on the TCC.
3. *Verification efficiency*. The overhead for the client should be constant, independently from the code base—i.e., a fixed number of hashes and digital signatures.
4. *Communication efficiency*. The protocol should be "non-interactive", requiring only a small additional constant amount of traffic to enable successful client verification.

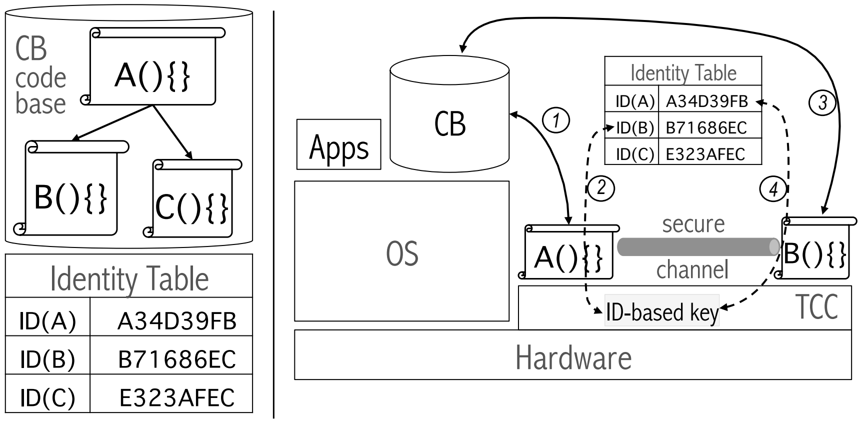


Figure 3.3: Sketch of our solution.

5. *TCC agnostic execution*. The protocol should use any underlying TC architecture as a black-box, thus allowing to retrofit existing trusted components.

### 3.1.4 Overview of our Solution

The core of our solution is displayed in Figure 3.3. On the left, the code base is depicted as a set of logically connected modules (arrows express the control flow graph) stored on the cloud provider's platform, working together to provide a service. Any two connected modules (e.g., *A* and *B*) communicate by means of authenticated messages, whose security is based on their identities (or hashes, e.g., *ID(A)*, *ID(B)*). The modules however do not contain any identity, instead, they contain the row number in the Identity Table to retrieve the identity they need (e.g., *A* requires *ID(B)* for sending an authenticated message to *B*, and similarly *B* requires *ID(A)* for receiving a message from *A*). The Identity Table thus contains the identities of all modules in the code base. On the right, the modules of our code base (CB) are initially in the untrusted environment, where there may be other untrusted applications (Apps) running above the untrusted OS.

Our protocol leverages the trusted component to execute the modules in isolation as follows. It starts the execution on the TCC of ① the modules of the code base, one by one, that are necessary to deliver the requested service (e.g., module *C* is not loaded). Given a particular client request, only the modules required to serve it are considered active (*A* and *B* in the figure). Active modules are loaded and run according to the correct execution order. Each module secures ② the intermediate state before it terminates. The next active module is then executed ③ and it validates the previous intermediate state ④. Such state is passed between modules by means of logical secure channels.

In contrast to having a single identity assigned to the code base, each module has its own identity. This identity is calculated as the hash of the code, thus following the usual practice. So it allows us to maintain backward compatibility, since we do not need to modify how the underlying TCC identifies the code, to achieve a general solution that is implementable on the trusted components available today. Our protocol builds a robust execution chain based on the identities of the modules, and guarantees that the modules are executed in the correct order with respect to the control flow graph—otherwise they are unable to validate the intermediate results when exchanged on the secure channels.

Each executing active module has access to data and resources required for the computation. Before the execution, each module is expected to receive either some input from the client (e.g., a request) or some intermediate state from other modules. Similarly, when it terminates, each module is expected to produce either an output for the client (e.g., a reply) or some intermediate state to be processed by the next module in the execution flow. Before and after the execution, every piece of data to/from any module is handled in the untrusted environment of the cloud provider's platform. Consequently, such data must be secured by means of the available TCC resources (e.g., secure storage).

An executing active module has access to the Identity Table (or identity set) of the code base. The module can use an identity in this table to request the TCC to secure data for (or to validate data received from) another module. In particular, intermediate states are transferred between modules through logical identity-dependent secure channels, whose security is enforced by the TCC. Such channels are "logical" in that the data is transferred between modules by the untrusted code on the platform (i.e., *A* releases data to the untrusted code, then upon termination, *A* is unloaded, so *B* is loaded and receives data from the untrusted code). The channel is secure as it guarantees (1) data integrity, through message authentication codes, and (2) the authentication of the end points based on their execution order (i.e., *A* sends to *B* and *B* receives from *A*, but *B* and *C* will never exchange data). A benefit of these channels is that they maintain all the data locally (at the cloud provider side) thereby avoiding the interaction with the client during the execution.

A client reply is authenticated through a TCC attestation, or through previously established symmetric secret keys. The last executed module (in the control flow graph) calls the TCC attestation service, and the TCC attests the module's identity. Such last module includes (the integrity measurement of) some parameters in its attestation, such as the client's initial request, the identity set of the code modules, and the reply. The client receives and verifies the attestation which, jointly with the parameters used to generate it, represents the proof of execution. By verifying the module's identity and the identity set, the client can trust that the code base correctly served the request.

Note that, by design, the hash chain created by the protocol enforces the execution order of the modules and guarantees their integrity by letting the TCC compute their identities. This means that the client does not have to be aware of the execution order for any specific execution, which is a highly desirable feature, nor has to verify the identity of any executed module except for the last one. This makes the protocol verification efficient.

## 3.2 Model

We extend the model introduced in Section 1.3.1 with some additional assumptions that are specific to the presented technique.

*Code base.* Our service code is composed by $q$ modules (or PALs[2]) $p_1, \ldots, p_q$. The control flow is a directed graph over the PALs describing their execution order. An execution flow is a sequence of PALs of finite but unknown length $n$ (e.g., $p_1, p_3, \ldots, p_4$) that respects the control flow. We refer to a generic execution flow as $m_1, \ldots, m_n$.

We assume that the service is either originally created or suitably partitioned into code modules by the service authors. We briefly discuss how such modules can be defined in Section 3.6.

*Provider-side.* The code base is available on the cloud provider's platform. Also, we leverage our TCC abstraction (Section 2.3.1), and in particular the execute, attest, verify, auth_put, auth_get and get_cert primitives.

*Client-side.* The client knows the cryptographic hashes of the attested PALs, and also the hash of the Identity Table (Figure 3.3), which represents the identity set. Ideally, the information is provided by the (trusted) authors of the code base and it requires a constant amount of space.

Also, the client knows and trusts the TCC's public key $K^{+}_{\text{TCC}}$, that is used to verify the attestations issued with the TCC's private key $K^{-}_{\text{TCC}}$. It can be obtained through an initial TCC Verification Phase: the client interacts with the cloud provider to retrieve $K^{+}_{\text{TCC}}$ and the associated certificate. If the public key is correctly certified by a trusted Certification Authority (e.g., the TCC manufacturer), then it can be trusted and used for verification. This step thus assumes a suitable implementation of the get_cert primitive (Section 2.3.1).

In addition, to simplify the description of a XMHF-TrustVisor-based implementation, we assume that $K^{+}_{\text{TCC}}$ is the hypervisor's public attestation key. Hence, we assume the client has al-

[2] Piece of Application Logic, using the notation of previous works [52, 53].

ready verified the hypervisor's identity and its public key, so we do not supply the hypervisor's identity and the TPM attestation to the verify primitive.

## 3.3 Secure Identification of Actively Executed Code

### 3.3.1 A Naive Solution

A client could verify and establish trust in the execution of a large code base by iteratively checking that each PAL is run correctly and respects the control flow graph. A relatively simple protocol to achieve this is the following: the client sends a request to the cloud provider to execute the first PAL $m_1$ on the TCC, and gives the input values for the service. When the PAL terminates, the cloud provider forwards to the client an attestation returned by $m_1$ that covers its identity (i.e., the hash of the module), the input and the output data. The output includes the identity of the next PAL to be run, besides the result of the execution (i.e., the intermediate state). Using the verify primitive (Section 2.3.1), the client can verify that the output is valid, since it was calculated with the correct code and the proper input. The same procedure can then be repeated for each PAL in the execution flow until the final result (i.e., the actual reply for the client) is produced by $m_{\mathrm{n}}$. The protocol ensures that the PALs are called in the right order and run over the correct data. Hence, it offers the required correctness guarantees.

Although the naive approach is secure and only attests the code modules that are actually executed, it has drawbacks. First, attestations are expensive, so a large number of executed modules can consume lots of TCC resources. Second, the approach is interactive, since it requires the client to verify each PAL and to mediate the transfer of the intermediate state between the executions of two modules. So it is not verification efficient as the client has to check every attestation and all the intermediate results produced by the executed PALs.

In the rest of the section we eliminate the above drawbacks. We explain a set of orthogonal techniques that: remove the interactivity with the client and reduce the TCC attestations to one (Section 3.3.2), address an issue with identity-based secure storage (Section 3.3.3), and optimize performance with a novel TCC-based key sharing solution (Section 3.3.4). Finally, we present a flexible and verifiable trusted execution protocol (Section 3.3.5).

### 3.3.2 Reducing Communication

When a trusted execution is requested, the client is only interested in obtaining the final reply (generated by the last executed module $m_{\mathrm{n}}$) and in verifying the validity of the whole execution (i.e., as if the code were executed as one single module). As a consequence, the intermediate states do not have to be transmitted to the client as long as the client can later check that they were handled correctly. Similarly, each PAL execution does not need to be attested as long as the client is still able to verify the correctness of the final result.

In the naive protocol, the client is involved in each PAL execution to ensure that the result of a piece of code $m_{\mathrm{i}}$ is properly provided as input to the next module $m_{\mathrm{i+1}}$. This is accomplished with two attestations returned to the client. The first is generated by $m_{\mathrm{i}}$ and provides evidence about $m_{\mathrm{i}}$'s intermediate state and the identity of the PAL that should be run next. The second is generated by $m_{\mathrm{i+1}}$ and provides evidence that the PAL received the correct intermediate state. Therefore, if a malicious cloud provider tampers with the execution, e.g., by running $m_{\mathrm{i+1}}$ with some incorrect input data, this can be detected with the second attestation. Hence, the verification of these attestations confirms that the intermediate state was correctly transferred from $m_{\mathrm{i}}$ to $m_{\mathrm{i+1}}$.

Attestations are a key mechanism in secure code execution but the overhead they impose is a concern. Attestations are essential because they convey the execution integrity property to a client. However, they are expensive since they involve digital signatures. In addition, each one imposes a non-zero overhead on the client. Verification requires not only the signature check, but also access to a copy of all data that is attested (i.e., at least the measurement of the code, the input and the output data).

In our approach, we build a "secure channel" between PALs without the client's supervision, thus saving attestations, communications and verification effort. We leverage the TCC secure storage capabilities (Section 2.3) to protect the data while it is saved locally in the cloud provider's untrusted storage. Recall that the TCC secure storage uses code identity to authenticate the auth_put and auth_get operations. By ensuring that only the correct code modules access security-sensitive intermediate state results, secure storage can be used as the basis to build a secure data transmission between PALs, instead of relying on attestations. Essentially, a *mutually-authenticated* channel is created: a PAL $m_{\mathrm{i}}$ authenticates the identity of the previous sender $m_{\mathrm{i-1}}$ when it gets the data from a protected input, and uses the identity of the next recipient $m_{\mathrm{i+1}}$ to securely store its results, before releasing them to the cloud provider's untrusted environment. It is because of this construction that the client only needs to verify the last executed PAL. Consequently, it is critical to ensure that such single verification can indeed be utilized to bootstrap trust into an arbitrary number of correct (though unverified) PALs that were called previously. We now present the end-to-end scenario for completeness and clarity.

The client first issues a service request and provides the input value $in$ and a fresh nonce $N$ to the cloud provider. The cloud provider calls the first module with the input: $\mathsf{execute}(m_1, in||N)$. $m_1$ carries out the initial part of the service computation and it invokes $\mathsf{auth\_put}(\mathsf{h}(m_2), \mathsf{h}(in)||N||out)$ before terminating. In other words, it saves a measurement of the input and any output intermediate state in secure storage, specifying the identity of the *only* subsequent PAL that is allowed to retrieve it (i.e., $m_2$ in the service execution flow). The outcome of the call is the protected data $\{\mathsf{h}(in)||N||out\}_K^{m_1-m_2}$, which is then returned to the cloud provider. Notice that $m_1$ is not attested, so it will not be verified by the client. The notation $\{\}_K^{m_i-m_j}$ refers to a message sent by $m_i$ to $m_j$ and protected with a TCC-internal secret key $K$.

The cloud provider next calls $\mathsf{execute}(m_2, \{\mathsf{h}(in)||N||out\}_K^{m_1-m_2})$ to run module $m_2$. The PAL authenticates the received data to make sure that it came from trusted source. This is achieved by calling $\mathsf{auth\_get}(\mathsf{h}(m_1), \{\mathsf{h}(in)||N||out\}_K^{m_1-m_2})$ with $m_1$'s identity. If the identity is not correct then auth_get fails, otherwise it succeeds and the PAL continues (the second part of) the service execution. Before it terminates, the PAL performs $\mathsf{auth\_put}(\mathsf{h}(m_3), \mathsf{h}(in)||N||out)$ to secure the new output intermediate state for the subsequent PAL. This procedure is repeated by all intermediate PALs.

The last PAL is attested and verified by the client. After $m_n$ retrieves the result from $m_{n-1}$ and runs the service code, it calls the attestation primitive $\mathsf{attest}(N, \mathsf{h}(in)||\mathsf{h}(out))$ to get a proof of execution that covers the input and output measurements, besides $m_n$'s own code. Since the attestation includes the nonce $N$, it also gives assurance about the freshness of the computation. The output with the attestation and the reply data $\{report, out\}$ is first released to the cloud provider's untrusted environment, and then forwarded to the client. The client verifies the execution proof by calling $\mathsf{verify}(m_n, \mathsf{h}(in), \mathsf{h}(out), N, K^+_{\mathrm{TCC}}, report)$ and accepts the result only if the primitive succeeds.

**Analysis.** The attestation binds together the initial inputs, the output and the identity of the last PAL. The cryptographic mutually authenticated chain that links $m_n$ to the previous PALs ensures that computation is performed *only among correct PALs*: when the verification of the correct execution of $m_n$ succeeds then, by construction, the client also trusts that $m_n$ can only have received data from a valid $m_{n-1}$; the same reasoning can be repeated up to $m_1$, which is the single entry point to the service and is the PAL that received the initial input data. Hence, correct intermediate PALs only accept data from (respectively deliver data to) correct PALs. Furthermore, as each piece of code specifies the receiver in auth_put, the overall execution order must match a valid control flow.

The only data that is accepted inside the trusted environment without being initially validated is the client's input. Similarly, the only data that is released outside without being protected in secure storage is the final output. However, their measurements are included in the last attestation, which allows the verification of the overall execution chain.

Freshness is guaranteed by the client provided nonce, which is propagated throughout the full execution. This prevents attacks where a malicious cloud provider would replace the output of a PAL $m_i$ with a value returned by the same PAL in a previous run of the protocol. Notice however that this attack could only succeed if the initial client input values (and so $\mathsf{h}(in)$) were the same in both service executions.

### 3.3.3 Addressing Looping PALs

PALs that exchange their intermediate state through TCC-based secure storage must have access to each others' identities. In auth_put, a module must specify the identity of the next PAL that should be granted access to secured data. Similarly, in auth_get, a PAL must give the identity of the sender module from which it is supposed to receive the data.

A straightforward approach is to include the identities of the next PALs statically in the code of a PAL. Unfortunately, this solution does not work out due to possible loops in the control flow graph of the service that end up creating unsolvable hash loops. Consider the example in Figure 3.4 (left-side), where a PAL contains some code and the identites of other PALs appended to its code, e.g., $m_1 = c_1||\mathsf{h}(m_3)$. It follows that:

$$\begin{aligned} m_1 &= c_1||\mathsf{h}(m_3) & &= c_1||\mathsf{h}(c_3||\mathsf{h}(m_1)||\mathsf{h}(m_4)) \\ m_3 &= c_3||\mathsf{h}(m_1)||\mathsf{h}(m_4) & & \end{aligned}$$

This example shows that loops in a control flow graph require a module to depend on a hash of itself. Solving the above equations cannot be done for cryptographic hashes as it would require us to violate the properties of these functions. Trapdoor functions could be used instead but they bring a few drawbacks. First, they are typically based on asymmetric cryptography. Hence, they are comparatively more expensive and further introduce the difficulty of protecting the private key. Second, they do not answer the more fundamental question of whether these hash loops can be avoided. In the following we show how to solve this issue without trapdoor functions.

Our approach uses a *level of indirection* to separate a PAL's code from its identity. We replace the critical identity information, hard-coded inside a PAL, with a lookup operation in a table Tab. The table contains the set of all PALs' identities and is built when the modules are originally

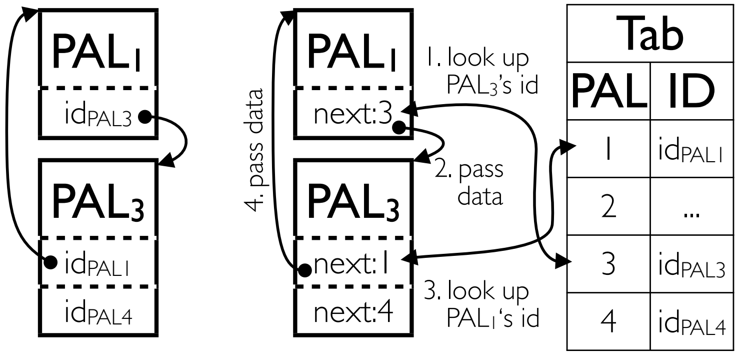


Figure 3.4: The *looping PALs problem* (left-side) and our solution of detaching PALs from identities (right-side).

created. In addition, we hard-code inside each PAL the index(es) in Tab of the correct next PAL(s) in the control flow (Figure 3.4 right-side). It is not necessary to hard-code the index(es) in Tab of the correct previous PAL(s) in the control flow, since the index can be provided at runtime as untrusted input—if such index is not the intended one, the PAL ends up retrieving the wrong identity from the table and therefore it is unable to authenticate the data that was destined to it by the intended sender PAL. The identities thus become independent from each other and each PAL's hash can be computed despite any loop in the control flow graph. The chain that binds our PALs together is now based on Tab, which translates an index into an identity.

Tab is critical to ensure the correct execution flow of the PALs. Hence, it has to be protected throughout the computation of all modules and eventually verified as follows. The first PAL accepts the table as input and propagates it to subsequent PALs using the TCC-based secure storage. The attestation of one PAL—the last executed in the control flow—has to cover the measurement of Tab. In order to eventually verify the execution, the client needs to be aware of: the last executed PAL's identity and the integrity measurement of Tab. Notice that this imposes only a small additional constant space and time overhead for any trusted execution.

The service developers should produce Tab together with the executable modules. Tab and PALs should be deployed on the cloud provider. The integrity measurements of (attested) PALs and Tab should be provided to the client to enable verification.

**Analysis.** The table Tab fixes the set of identities of the PALs that are allowed to implement each part of the service functionality. When the client verifies the correctness of the last executed PAL, say $m_n$, together with Tab, the client can trust that only valid PALs were used throughout the execution process. In fact, Tab ensures that only correct identities are used for secure storage operations, and only correct PALs have access to securely stored data that is critical for the

$$K_{sndr-rcpt} = \begin{cases} f_K(\textsc{Reg}, rcpt) & \text{on kget\_sndr by } sndr \\ f_K(sndr, \textsc{Reg}) & \text{on kget\_rcpt by } rcpt \end{cases}$$

Figure 3.5: Identity-dependent key derivation construction for Secure Storage. $rcpt$ is the identity of the recipient PAL and $sndr$ is the identity of the sending PAL. REG is the register inside the TCC that stores the identity of the currently executing PAL. It is equivalent to a $PCR$ on TPMs [76] or to the $MRENCLAVE$ register in Intel SGX [107]. $f()$ is a keyed hash function.

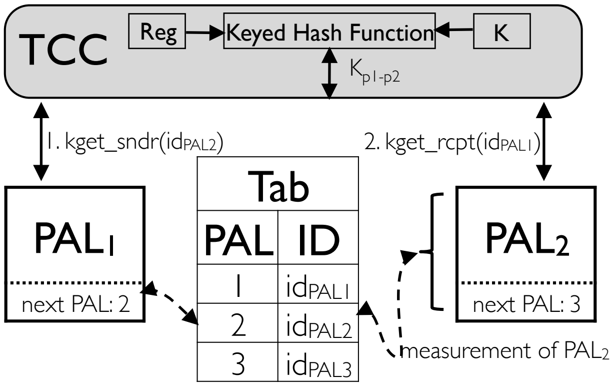
execution.

### 3.3.4 Novel Secure Storage Solution

Secure channels that are used to transfer intermediate results across trusted PAL executions should be available with low overhead. In particular, they should be (1) fast to set up, (2) require minimal TCC support, and (3) ensure mutual authentication of the end points.

The secure channel design described above can be built on current trusted components, but the core construction inside its implementation is usually inefficient as it provides more guarantees than desired. For example, on TPMs $v1.2$, sealed storage is based on asymmetric cryptography, which provides non-repudiation unnecessarily. As another example, while symmetric algorithms are available on TPMs $v2.0$, the trusted component still implements and enforces data access control (i.e., it checks whether the identified code is allowed to access the data), besides guaranteeing the confidentiality or integrity of sealed data. Intel SGX instead uses a different paradigm. The `ENCLU[EGETKEY]` instruction (for sealing) only provides a key to the Enclave based on its identity. The key is used by the Enclave to protect the data that can be then released outside its secure execution environment. Unfortunately, when two Enclaves need a shared secret key, they have to run an authentication protocol [98] to bootstrap trust in each other's attestations and validate public Diffie-Hellman keys. This involves at least two message exchanges, besides asymmetric cryptographic operations.

We propose a new construction that binds a secret shared key to the identities of two PALs efficiently. In particular, for any two PALs, the construction can build a secret key to create their secure channel. For any such key, only the PALs with the correct identity can access it. Keys are derived from a master key $K$, which is a secret symmetric key that the TCC maintains internally for computing identity-dependent keys. Any PAL $m_i$ can use an identity-dependent key to protect data. Any such key depends on: $K$, $m_i$'s identity and another PAL $m_j$'s identity. Also, the key can only be accessed by $m_i$ and $m_j$. When module $m_i$ wants to secure some information, it calls kget_sndr with the identity of the receiver PAL $m_j$. The TCC then performs the operations in Figure 3.5 to derive a secret shared key $K_{m_\mathrm{i}-m_\mathrm{j}}$ that is then returned to the PAL. To retrieve the same key later, $m_\mathrm{j}$ invokes kget_rcpt to perform an equivalent operation. Provided that the source and the recipient PALs respectively supply each other's identity (i.e., resp. $rcpt\equiv\mathsf{h}(m_\mathrm{j})$ or $sndr\equiv\mathsf{h}(m_\mathrm{i})$) to the TCC, the computed key is the same in the two cases.



| PAL | ID |
|---|---|
| 1 | id$_{PAL1}$ |
| 2 | id$_{PAL2}$ |
| 3 | id$_{PAL3}$ |

Figure 3.6: Identity-based Secure Storage. It enables two PALs to share a mutually authenticated secret key in zero rounds, with no message exchange. Two PALs can use such key to transfer data securely with minimal overhead.

The usage of the shared key for the new secure storage construction is shown in Figure 3.6. It allows the protection of data that is transmitted between adjacent PALs. PALs can use the identity table Tab to look up the identity of the next executing PAL according to the control flow. Next, the sender PAL (resp. recipient PAL) calls kget_sndr (resp. kget_rcpt) to obtain the shared key. The key is then used by the PAL to secure (resp. validate) the data to be released to (resp. supplied by) the cloud provider. As we will detail later, we wrap this functionality inside the auth_put and auth_get TCC primitives.

Our construction can be seen as a generalization of the Intel SGX approach, since a PAL is allowed to set up a secure channel not only with another code module but also with itself—e.g., to seal and save data in external untrusted storage. Instead, in contrast to TPM sealed storage, in our construction the TCC does not make any access control decision on whether to accept or reject a PAL request, based on its current configuration (e.g., the value of the PCR registers) and the information included in the sealed data. The TCC always generates symmetric keys on-demand. It is up to a PAL to decide whether to encrypt or just authenticate some data, and what PAL is the recipient of the data. So it is crucial that correct modules have access to correct identities. If data is encrypted and an invalid module attempts to decrypt it, it simply gets some random information because with high probability the wrong key is used. Similarly, if a valid module is run with incorrect data (possibly from a malicious module), it is simply unable to authenticate the input.

**Analysis.** In the key derivation function, the TCC uses the computed identity of the currently executing PAL and a possibly untrusted identity provided by the PAL itself. These are positioned differently by the TCC in the $f$ function, depending on whether the current PAL is saving or retrieving data (as in Figure 3.5). The presence and the eventual verification of table Tab ensure that only correct identities (and thus PALs) are used to call the key derivation function. Furthermore, since a valid PAL forwards the data to the intended next PAL in accordance with the control flow, this guarantees that the right order of execution is followed and that only the correct next PAL can decrypt/validate the data.

**How the TCC primitives are extended.**

The implementation of the auth_put primitive (previously presented in Algorithm 7 and Algorithm 8) can be extended to include our construction as highlighted in Algorithm 19 and Algorithm 20 (in dark background, bottom-side). The construction allows two pieces of code to retrieve (line 1) an identity-dependent secret key (based on both their identities) that they share immediately without exchanging messages. The construction is efficient as it involves computing just a single hash. We point out however that while this has been evaluated in XMHF-TrustVisor (Section 3.4.3), because the hypervisor can be easily customized, it instead requires small (in our opinion) extensions to the CPU microcode to be supported by SGX.

The implementation of the auth_get primitive (previously presented in Algorithm 9 and Algorithm 10) can be extended to include our construction as highlighted in Algorithm 21 and Algorithm 22 (in dark background, bottom-side). As the implementation mirrors the operations in the auth_put primitive, we only highlight the simplicity of implementing it in XMHF-TrustVisor, and the algorithmically visible benefits—i.e., no key exchange, no asymmetric cryptography, and shared secret key available with just one instruction—if it were implementable on SGX.

### 3.3.5 A Flexible Trusted Execution Protocol

We now integrate the presented techniques into the *Flexible and Verifiable Trusted Execution (fvTE)* protocol detailed in Figure 3.7. The protocol is based on our TCC abstraction (Section 2.3, enhanced with our new constructions in Section 3.3.4) and ensures all properties specified in Section 3.1, namely it allows a client to securely and efficiently check the correctness of an arbitrary code execution. We now describe the main steps.

The client begins the protocol by submitting a service request to the cloud provider, including the service input $in$ and a nonce $N$. The cloud provider then starts running the first PAL $m_1$

```
Input: (recipient) code identity, data
Output: secured data
trigger hyper-call for sealing data specifying the recipient identity[a]
return secured data
```

```
Input: (recipient) code identity, data
Output: secured data
trigger hyper-call to retrieve the identity-dependent sender key $K_{\mu PCR[0]-recipient\ identity}$
use key to authenticate/encrypt data
return secured data
```

**Algorithm 19:** auth_put primitive for XMHF-TrustVisor. Above, the original implementation from Algorithm 7. Below, dark areas highlight the differences of the implementation using our construction.

```
Input: (recipient) code identity, data
Output: secured data
    (following [98])
if there already exists a sealed public/private Diffie-Hellman key pair then
    retrieve it from shared memory and unseal it using EGETKEY
else
    generate a public/private Diffie-Hellman key pair
    seal it using EGETKEY and save it in shared memory
end if
run EREPORT to produce a local attestation of the public key for the recipient code identity
send attestation through shared memory
retrieve the recipient's local attestation of its key, and its key, from shared memory
run EGETKEY to retrieve the report key
verify that the local attestation belongs to the intended recipient and that it attests the received key
compute the secret key shared with the recipient code
use key to authenticate/encrypt data
return secured data
```

```
Input: (recipient) code identity, data
Output: secured data
run EGETKEY to retrieve the identity-dependent sender key $K_{\texttt{MRENCLAVE}-recipient\ identity}$[b]
use key to authenticate/encrypt data
return secured data
```

**Algorithm 20:** auth_put primitive for Intel SGX. Above, the original implementation from Algorithm 8. Below, dark areas highlight the differences of the implementation using our construction.

[a] by suitably setting a $digestAtRelease$ value that allows unsealing the data only when $\mu PCR[0]$ (i.e., the identity of the running code) will match the recipient code identity.

[b] Note. This is our novel proposal for secure message passing between enclaves. At the time of writing, this construction is not available on SGX so it is not implementable.

```
Input: (sender) code identity, secured data
Output: validated data or $\perp$
trigger hyper-call for unsealing data [a]
if data has been unsealed then
    use sender code identity to compute $expected\_digestAtCreation$
    if $digestAtCreation$ matches $expected\_digestAtCreation$ then
        return data
    end if
end if
return $\perp$
```

[a] The hyper-call decrypts the data blob and checks that the currently executing code is authorized to access it—in our case by using $\mu PCR[0]$ to check the $digestAtRelease$ in the data. If (and only if) so, it returns the data and a $digestAtCreation$ value that the running code can use to check the identity of the code that originally sealed the data.

```
Input: (sender) code identity, secured data
Output: validated data or $\perp$
trigger hyper-call to retrieve the identity-dependent receiver key $K_{sender\ identity-\mu PCR[0]}$
use key to validate/decrypt data
if data is valid then
    return data
end if
return $\perp$
```

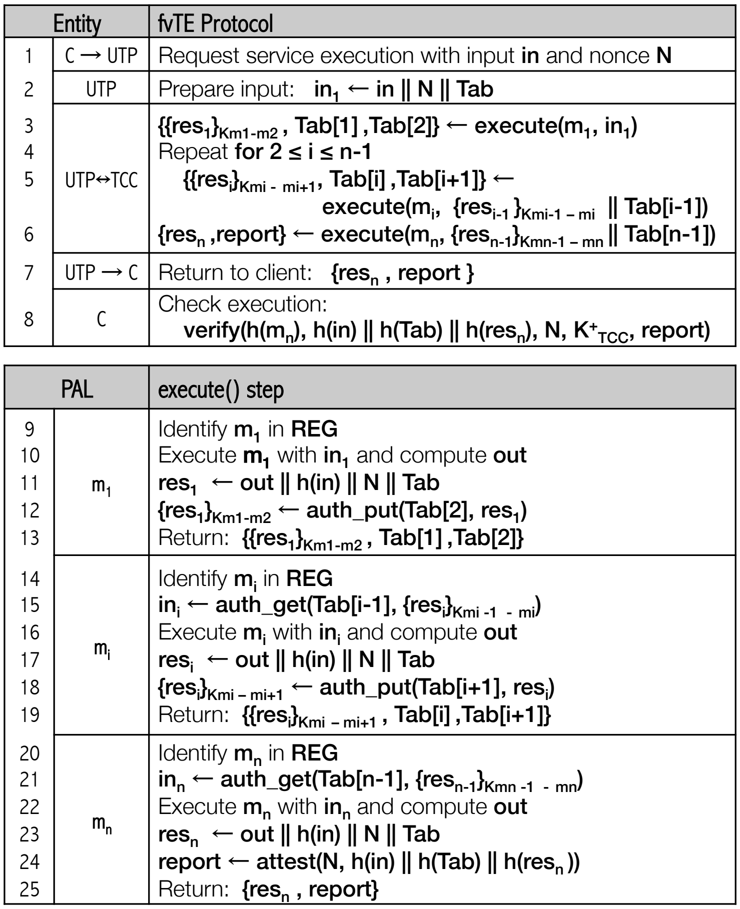
**Algorithm 21:** auth_get primitive for XMHF-TrustVisor. Above, the original implementation from Algorithm 9. Below, dark areas highlight the differences of the implementation using our construction.

```
Input: (sender) code identity, secured data
Output: validated data or $\perp$
(following [98])
if there already exists a sealed public/private Diffie-Hellman key pair then
    retrieve it from shared memory and unseal it using EGETKEY
else
    generate a public/private Diffie-Hellman key pair
    seal it using EGETKEY and save it in shared memory
end if
run EREPORT to produce local attestation of the public key for the sender code identity
send attestation through shared memory
retrieve the sender's local attestation of its key, and its key, from shared memory
run EGETKEY to retrieve the report key
verify that the local attestation belongs to the intended sender and that it attests the received key
compute the secret key shared with the sender code
use key to validate/decrypt data
if data is valid then
    return data
else
    return $\perp$
end if
```

```
Input: (sender) code identity, secured data
Output: validated data or $\perp$
run EGETKEY to retrieve the identity-dependent recipient key $K_{sender\ identity-\texttt{MRENCLAVE}}$ [b]
use key to validate/decrypt data
if data is valid then
    return data
else
    return $\perp$
end if
```

**Algorithm 22:** auth_get primitive for Intel SGX. Above, the original implementation from Algorithm 10. Below, dark areas highlight the differences of the implementation using our construction.

[b] Note. This is our novel proposal for secure message passing between enclaves. At the time of writing, this construction is not available on SGX so it is not implementable.



| Entity | | fvTE Protocol |
|---|---|---|
| 1 | C → UTP | Request service execution with input **in** and nonce **N** |
| 2 | UTP | Prepare input: $in_1 \leftarrow in \parallel N \parallel Tab$ |
| 3<br>4<br>5<br><br>6 | UTP↔TCC | $\{\{res_1\}_{Km1\text{-}m2}, Tab[1], Tab[2]\} \leftarrow execute(m_1, in_1)$<br>Repeat **for $2 \leq i \leq n\text{-}1$**<br>$\{\{res_i\}_{Kmi\ \text{-}\ mi+1}, Tab[i], Tab[i+1]\} \leftarrow$<br>$execute(m_i, \{res_{i\text{-}1}\}_{Kmi\text{-}1\ –\ mi} \parallel Tab[i\text{-}1])$<br>$\{res_n, report\} \leftarrow execute(m_n, \{res_{n\text{-}1}\}_{Kmn\text{-}1\ –\ mn} \parallel Tab[n\text{-}1])$ |
| 7 | UTP → C | Return to client: $\{res_n, report\}$ |
| 8 | C | Check execution:<br>$verify(h(m_n), h(in) \parallel h(Tab) \parallel h(res_n), N, K^{+}_{TCC}, report)$ |

| PAL | | execute() step |
|---|---|---|
| 9<br>10<br>11<br>12<br>13 | $m_1$ | Identify $m_1$ in **REG**<br>Execute $m_1$ with $in_1$ and compute **out**<br>$res_1 \leftarrow out \parallel h(in) \parallel N \parallel Tab$<br>$\{res_1\}_{Km1\text{-}m2} \leftarrow auth\_put(Tab[2], res_1)$<br>Return: $\{\{res_1\}_{Km1\text{-}m2}, Tab[1], Tab[2]\}$ |
| 14<br>15<br>16<br>17<br>18<br>19 | $m_i$ | Identify $m_i$ in **REG**<br>$in_i \leftarrow auth\_get(Tab[i\text{-}1], \{res_i\}_{Kmi\ \text{-}1\ \ \text{-}\ mi})$<br>Execute $m_i$ with $in_i$ and compute **out**<br>$res_i \leftarrow out \parallel h(in) \parallel N \parallel Tab$<br>$\{res_i\}_{Kmi\ –\ mi+1} \leftarrow auth\_put(Tab[i+1], res_i)$<br>Return: $\{\{res_i\}_{Kmi\ –\ mi+1}, Tab[i], Tab[i+1]\}$ |
| 20<br>21<br>22<br>23<br>24<br>25 | $m_n$ | Identify $m_n$ in **REG**<br>$in_n \leftarrow auth\_get(Tab[n\text{-}1], \{res_{n\text{-}1}\}_{Kmn\ \text{-}1\ \ \text{-}\ mn})$<br>Execute $m_n$ with $in_n$ and compute **out**<br>$res_n \leftarrow out \parallel h(in) \parallel N \parallel Tab$<br>$report \leftarrow attest(N, h(in) \parallel h(Tab) \parallel h(res_n))$<br>Return: $\{res_n, report\}$ |

Figure 3.7: fvTE protocol run by client C, the trusted component TCC and the cloud provider UTP (above, lines 1-8), and the execute step at the various PALs $m_i$ (below, lines 9-25). A single attestation and verification allows the client to trust the service execution, independently from the number of executed PALs. Also, only PALs that are necessary to serve a specific request are loaded and executed using the TCC.

by providing the client's input, the nonce and the identity table, i.e., $< in||N||\mathsf{Tab} >$ (lines 2-3). Notice that this is the only entry point of non-authenticated (and thus untrusted) data. However, the correctness of such data will be eventually verified by the client before accepting the reply.

The first PAL is run with the input and produces an intermediate state *out* (lines 9-10). Before returning, it prepares the data to be forwarded to the next PAL: the output, a hash of the input, the nonce and the identity table (line 11). The hash is used as an optimization to minimize the information to be transferred to subsequent PALs. This data is secured using auth_put, specifying the identity of the PAL that should follow in the execution flow ($\mathsf{h}(m_2) \equiv \mathsf{Tab}[2]$ [3]). The PAL terminates by releasing to the cloud provider the secured intermediate state, and the identity of

[3] Notice that "2" actually corresponds to the index of the next PAL in the execution flow that is hard-coded in $m_1$. The index is used for the lookup operation in Tab. We use this simplification in the description for brevity.

the current and the next PAL (line 13).

The execution of the subsequent intermediate PALs proceeds similarly. They use auth_get to obtain the previous intermediate state (line 15), whose validity derives from the properties of the secure channel. They execute their service code and propagate the result according to the expected control flow (lines 17-19). Notice that the values $< \mathsf{h}(in)||N||\mathsf{Tab} >$ are simply left unchanged by each intermediate PAL as a way to propagate them to the final PAL ($\mathsf{h}(m_{\text{n}}) \equiv \mathsf{Tab}[n]$).

$m_{\text{n}}$ prepares the output for the client. After it retrieves the intermediate result from secure storage, it executes the code (lines 21-22) and performs an attestation that binds together $m_{\text{n}}$'s identity, the nonce, the client's request input, the identity table and the final output (lines 23-24). When $m_{\text{n}}$ terminates, it releases the final output and the report to the cloud provider (line 25).

The cloud provider forwards the output to the client for verification (line 7). At this point, the client has the following information: $m_{\text{n}}$'s identity and $\mathsf{h}(\mathsf{Tab})$ that were outsourced by the authors of the code; the originally created request ($in$) and the fresh nonce; the final output ($out$) and the attestation as issued by $m_{\text{n}}$; the trusted TCC's public key (see assumptions in Section 3.2). The client can thus verify the attestation, and so the execution correctness, and establish trust in the service output (line 8).

**Discussion.** The protocol ensures that the properties in Section 3.1.3 are achieved as follows:

1. *Secure proof of execution*. The proof is unforgeable because it is conveyed by an attestation, i.e., a digital signature over (secure hashes of) the input, the output, the identity table. The signature is linked to the TCC hardware root of trust through a chain of digital certificates, whose ultimate root is a Certification Authority trusted by the client. The proof is unambiguous because of the attested identity of the last PAL $m_{\text{n}}$ and Tab, and it is unique due to the inclusion of the nonce $N$. The execution flow cannot be tampered with, since only the correct PALs can be run in the expected order. This last point is ensured through the novel storage primitive (and the identity table Tab) that prevents invalid PALs from accessing and tampering with the output of the intermediate states.

2. *Verification efficiency*. The client only has to perform a constant number of hashes and check one digital signature to validate the result. Such verification effort is independent from the number of executed PALs.

3. *Communication efficiency*. The client interacts only once with the cloud provider to send the input and receive the output of the service. Also, the client provides and receives a constant additional amount of data, i.e., the nonce $N$ and $report$.

4. *Low TCC resource usage*. Throughout the protocol, only the PALs that are required to serve

the client request are loaded, identified and run. Furthermore, public key cryptography usage is limited to one attestation, while symmetric cryptography is used for fast key derivation on the TCC. Hence, the protocol consumes TCC resources efficiently and proportionally to what is actually executed.

5. *TCC-agnostic execution*. The execution protocol only uses the TCC abstraction (Section 2.3). As the interface can be implemented on different trusted components, the protocol is not restricted to any specific architecture, so it is general.

#### 3.3.5.1 Amortizing the attestation cost

Reducing the number of attestations provides benefits both to the cloud provider and to the client. However, producing one attestation for each request from a client is still computationally expensive and also imposes some verification overhead on the client. It is common practice to avoid this overhead by establishing a secure channel based on a symmetric secret key between the trusted environment and the client. In particular, one attestation is produced and verified to establish trust in a symmetric key. Then the client uses the key to issue multiple requests and to authenticate the associated replies. So the cost of one attestation can be amortized over multiple requests from a client.

We sketch how this technique can be used also with our protocol. The code base can be enriched with another PAL $m_{\text{C}}$ that establishes the secure channel with the client. $m_{\text{C}}$ receives the client's fresh public key $pk_C$ as input at the beginning of the computation. It assigns the identity $id_C = \mathsf{h}(pk_C)$ to the client; it uses kget_sndr (Section 3.3.4) to retrieve the identity-dependent key $K_{m_{\text{C}}-C}$ to be shared with the client; it encrypts $K_{m_{\text{C}}-C}$ with $pk_C$ and attests the result. The attestation and the encrypted data are sent back to the client. The client verifies the attestation and retrieves $K_{m_{\text{C}}-C}$. In subsequent requests, the client authenticates (or encrypts) messages with $K_{m_{\text{C}}-C}$ and attaches $id_C$. The client's identity allows $m_{\text{C}}$ to recompute $K_{m_{\text{C}}-C}$ without maintaining any state for the secure channel. $m_{\text{C}}$ can thus authenticate requests from the client and forward them to the first PAL in the original execution flow. Similarly, $m_{\text{C}}$ receives the computed reply from the last PAL (in the original execution flow) and authenticates it with $K_{m_{\text{C}}-C}$ for the client.

Notice that only $m_{\text{C}}$ is attested once for setting up the secure channel with the client, and the (encrypted) secret shared key is the attested output. Also, since Tab is included in the attestation (as mandated by our protocol), the client implicitly verifies the identities of the other PALs that later receive and process its request.

## 3.4 Experimental Analysis

This section focuses on the implementation and evaluation of our protocol when applied to a real-world service. The protocol is used to securely link together code modules of the widely-deployed SQLite database engine. We model the implementation using Scyther for automatic verification. Our results show that the code identification overhead can be significantly reduced without trading off security and functionality.

### 3.4.1 Implementation

*Trusted component.* We implemented the TCC using XMHF-TrustVisor [53, 94], which is based on a hardware TPM, and whose code is open-source and easy to customize. Some background about the hypervisor can be found in Section 2.2.1.

In order to implement our protocol, we modified XMHF-TrustVisor by adding three hyper-calls. The first makes memory available to a PAL in its address space. This avoids allocating memory in the untrusted environment, then transferring it to the trusted environment and making it accessible to a PAL as dynamic memory. Consequently, such memory space is neither part of a PAL's identity, nor of a PAL's input data, and it can be provided more efficiently. The second hyper-call is kget_sndr, which is used in the auth_put primitive to retrieve a shared key to secure some data for a known receiver PAL. As the TCC only computes a secret key, this allows a developer to choose and implement the security technique that is most suitable for the application (e.g., message authentication codes or authenticated encryption). The third hyper-call is kget_rcpt, which is used in the auth_get primitive to retrieve a shared key to validate some data that was previously secured by a known sender PAL. The TCC-specific key (i.e., $K$ in Section 3.3.4) that is used for identity-dependent key derivation is initialized inside XMHF-TrustVisor when the platform boots.

*Platform.* We used a Dell PowerEdge R420 Rack Server, with a 2.2GHz Intel Xeon E5-2407 CPU, 3GB of memory, a TPM v1.2, and running Ubuntu 12.04 with a Linux kernel 3.2.0-27. The resources were fully dedicated to our experiments.

*Application.* We apply our protocol to the SQLite database engine [188], which has a code base of about $88K$ lines of source code. SQLite is open-source and widely deployed, e.g., on Android [109], iCloud [110] and other operating systems [111].

We create a multi-PAL SQLite engine with a small per-operation code footprint. Different PALs handle specific queries. Each PAL is handcrafted by trimming the unused code off the original code base. Then, our protocol is used to securely link these PALs together.

Our current multi-PAL SQLite engine consists of $4$ PALs that implement some of the most representative SQL operations. We emphasize that additional operations can be included by following the same approach (see Section 3.6) so to match the functionality of the original database engine. $PAL_0$ is the first one called from the untrusted environment on the cloud provider's platform and it receives the input data from the client. $PAL_0$ parses the client's request to recognize the type of query, and then forwards it to a specialized PAL for the execution by means of our secure channels. $Select$ queries are passed to $PAL_{SEL}$. $Insert$ queries are sent to $PAL_{INS}$. $Delete$ queries are passed to $PAL_{DEL}$. The last executed PAL builds the reply that is released to the cloud provider's untrusted environment, from which it is then forwarded to the client. Therefore, requests from the client follow the execution flows: $PAL_0 \rightarrow PAL_{SEL}$, $PAL_0 \rightarrow PAL_{INS}$, or $PAL_0 \rightarrow PAL_{DEL}$.

We compare multi-PAL SQLite against a baseline implementation of the full SQLite database engine. We implemented it as a monolithic $PAL_{SQLITE}$ that can execute any query.

We perform end-to-end experiments, where a client performs $select$, $insert$ and $delete$ queries on the server that maintains a database. Queries are received through a ZeroMQ [112] socket at the cloud provider, and delivered to $PAL_0$ for initial processing. The experiments were based on a small size database because it highlights the overhead due to code identification, which is the focus of this chapter.

### 3.4.2 Automatic Verification

We verified the $fvTE$ protocol applied to SQLite using Scyther [100, 101], a public tool for the automatic verification of security protocols. We chose Scyther as it supports unbounded verification of security properties or their violation by providing feasible attacks.

*Protocol Modeling.* The security protocol is performed among the following entities: the client, the $4$ PALs and the TCC. The cloud provider is untrusted and it is modeled by Scyther as an adversary that is able to forge and replay messages. We describe the execution verification of a *select* query—it will be evident that it can be adapted to other executions in a straightforward manner.

Messages are exchanged on two channels: one between the client and the TCC; and another between the TCC and a PAL that is executing. The first one is modeled as an insecure channel because the client and the TCC do not share any secret. So the first message is not secured and the last message is signed by the TCC (i.e., attested through the private key $K^{-}_{\text{TCC}}$). The second channel is instead modeled as a secure channel. We let the TCC and each PAL share a fresh secret key (e.g., $K_{TCC \leftrightarrow PAL} \equiv K_{PAL \leftrightarrow TCC}$) to secure their communication. The reason is that each PAL runs (and terminates) above the TCC when the execution environment is already isolated. This implies a secure data/control transfer between the TCC and each PAL.

A logical secure channel is available between pairs of PALs. The channel is protected with the key (for instance, $K_{PAL_0 \leftrightarrow PAL_{SEL}}$) shared between the indicated PALs. The TCC essentially forwards messages between the direct channels that it establishes with each PAL. This is modeled through message encapsulation: a PAL first secures the message using the key that it shares with another PAL, and then it secures the message again using the key shared with the TCC. The security of the channel derives from our construction in Section 3.3.4.

**Protocol Verification.** The execution chain is verified in three steps. First, the TCC validates that $PAL_0$ successfully completes an execution on inputs $< in, N, Tab >$ and delivers a response $< res_{PAL_0} >$ securely linked to the inputs. This allows the TCC to trust that $res_{PAL_0}$ is the correct output of $PAL_0$. Second, the TCC validates that $PAL_{SEL}$ successfully completes an execution on input $< res_{PAL_0} >$, which includes $h(in), N, Tab$, and delivers a response $< res_{PAL_{SEL}} >$ securely linked to the inputs. This lets the TCC trust that $res_{PAL_{SEL}}$ is the correct output of $PAL_{SEL}$. Third, the client validates that the TCC successfully completes an execution on inputs $< in, N, Tab >$ and delivers a response $res_{PAL_{SEL}}$ securely linked to the inputs. Finally, this allows the client to trust that $res_{PAL_{SEL}}$ is the valid output.

Scyther verified the protocol execution in about 35 minutes, on a MacBook Pro with a 2.3GHz Intel i7 CPU.

**Discussion.** The reader should note that the successful verification refers to the $fvTE$ protocol as applied to the multi-PAL SQLite design and not to the general protocol (in Figure 3.7). However, this verification together with the analysis performed during the protocol description (Section 3.3.5) gives us confidence that our approach is correct. Verifying an actual implementation is an orthogonal problem that could be addressed with Ironclad Apps [113].

### 3.4.3 Evaluation

We evaluate the multi-PAL SQLite and compare it against the full monolithic SQLite. An always-positive speed-up was observed with our design, which shows that for this setting it is convenient to load and integrity-measure only the modules that are executed out of a large code base.

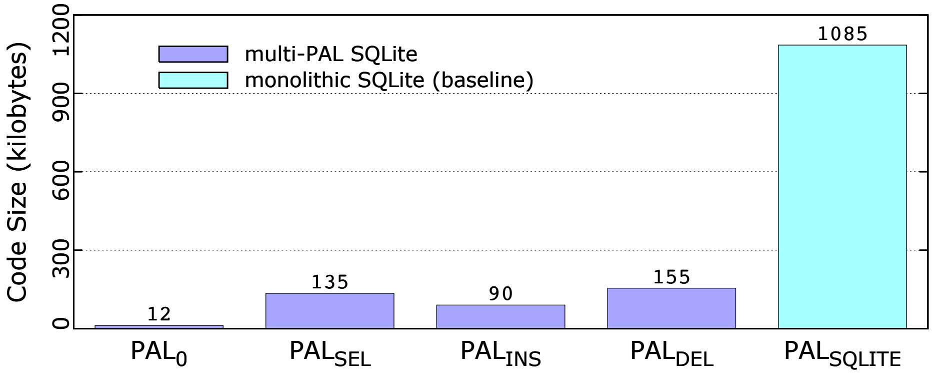


Figure 3.8: Size of each PAL's code in our SQLite code base.

#### 3.4.3.1 Code size

The size of the code for each PAL protected by XMHF-TrustVisor at registration time is shown in Figure 3.8. The size of the full SQLite implementation is about 1MB, while common operations such as $select$, $insert$, $delete$ can be implemented in as little as 9-15$\%$ of the code base.

#### 3.4.3.2 End-to-end performance

The performance results for each execution flow are displayed in Figure 3.9, and summarized in Table 3.1. Each run is one end-to-end query execution, i.e., the client sends one request and receives the corresponding reply. We have included the execution times with and without attestation. The average of at least 10 runs is displayed with the $95\%$ confidence interval. XMHF-TrustVisor computes an attestation using a $2048bit$ RSA key and, in our testbed, it takes around $56ms$. Such overhead could be reduced by establishing a secure session with the client (see Section 3.3.5).

| *speed-up* | W/ ATTESTATION | W/O ATTESTATION |
|---|---|---|
| INSERT | 1.46× | 2.14× |
| DELETE | 1.26× | 1.63× |
| SELECT | 1.32× | 1.73× |

Table 3.1: Summary of the achieved per-operation speed-up.

Overall, our protocol improves substantially on the previous approach. For example, $insert$ is about $1.46\times$ faster than the traditional approach using the monolithic SQLite; the result could be improved to become up to $2\times$ faster by considering more efficient attestation mechanisms. Notice that if the original code base gets larger, then the benefit increases.

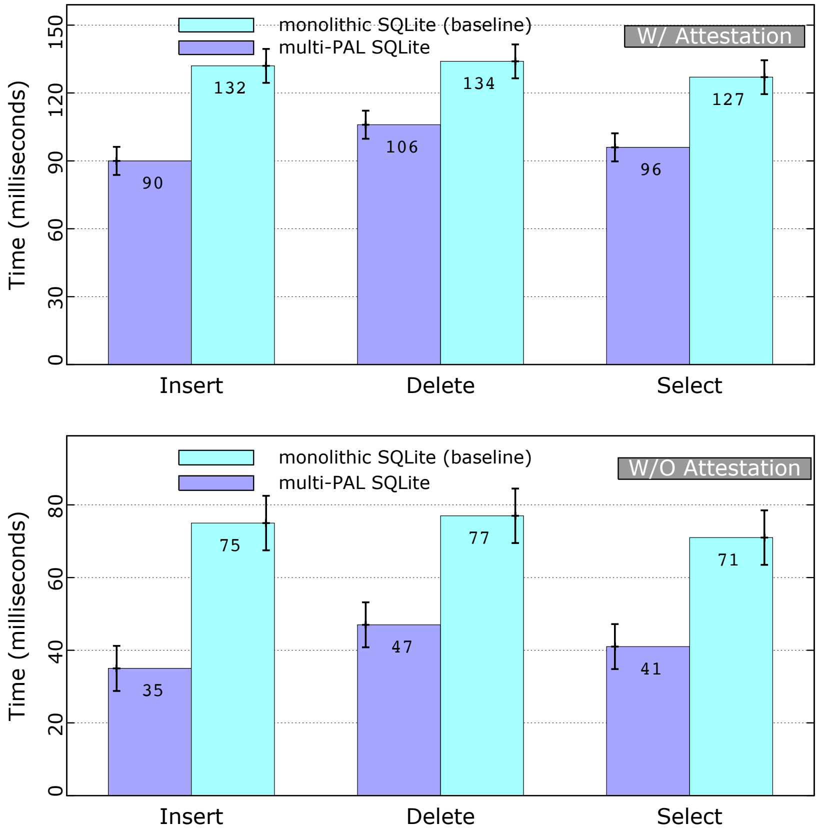


Figure 3.9: Performance comparison between the multi-PAL and the monolithic SQLite databases. (lower is better)

At the application level (i.e., without considering the underlying TCC overhead), the execution time of SQLite is similar for queries that are executed in the monolithic $PAL_{SQLITE}$ or in the small PALs. This is expected since they execute essentially the same code on the same state. Consequently, the performance differences are mainly the result of the different size of the code that is loaded in the trusted environment.

Finally, we measured the overhead of $PAL_0$ in our end-to-end experiments. $PAL_0$ terminates its execution in about $6ms$. Considering attestation, this corresponds to an overhead of $6.6\%$ for $insert$, $5.6\%$ for $delete$, $6.2\%$ for $select$. Without attestation, the overhead is $17.1\%$, $12.7\%$, $14.6\%$ respectively.

#### 3.4.3.3 Optimized vs. non-optimized secure channels

We compare our secure storage construction (Section 3.3.4) with the original one of XMHF-TrustVisor (i.e., *seal* and *unseal*). Both use symmetric cryptography, but XMHF-TrustVisor's secure storage requires more operations for: (i) managing TPM-like data structures because it implements a software micro-TPM; (ii) using AES for encryption, in order to guarantee secrecy of sealed data; (iii) retrieving random numbers for the initialization vector to guarantee semantic security; (iv) using SHA1-HMAC for integrity protection. Instead, our construction only uses SHA1-HMAC, keyed with the TCC secret created at boot time, to derive identity-dependent keys.

The results of the performance measured inside the hypervisor are: $15\mu s$ and $16\mu s$ for kget_rcpt and kget_sndr; and $122\mu s$ and $105\mu s$ for *seal* and *unseal* respectively. The operations in our construction are respectively $8\times$ and $6.5\times$ faster. In our experiments, using XMHF-TrustVisor's native secure storage (recall from Section 3.3.4 that both can be used to implement secure channels) does not change the results in Figure 3.9 noticeably. The difference in overhead is at least two orders of magnitude smaller than the end-to-end execution time. Notice however that in large-scale services of several interconnected PALs and long execution flows, such overhead could become non-negligible.

## 3.5 Performance Model for Code Identification

In this section we devise a performance model for code identification to study under what circumstances using the *fvTE* protocol outperforms the traditional approach of monolithic trusted executions. For the traditional approach, we can model the costs for code execution as follows:

$$T = \underbrace{(t_{is}(\mathcal{C}) + t_{id}(\mathcal{C}) + t_1)}_{\text{code protection cost}} + \underbrace{(t_{is}(in) + t_{id}(in) + t_2)}_{\text{input protection cost}} + \underbrace{\underbrace{(t_{is}(out) + t_{id}(out) + t_3)}_{\text{output protection cost}} + \underbrace{t_{att}}_{\text{attestation cost}}}_{\text{TCC-dependent costs}} + \underbrace{t_X}_{\text{execution cost}}$$

We distinguish between TCC-related costs and application-level costs. The latter ($t_X$) is invariant with respect to the trusted execution protocol actually used, and only depends on the platform where the application runs. The former instead depends on the TCC and on the implemented protocols for isolation ($is$), identification ($id$) and attestation ($att$) of a code base ($\mathcal{C}$) and input/output ($in/out$) data. As shown later, identification and isolation costs are linear in the size of the argument ($\mathcal{C}$, $in$, or $out$), while $t_1, t_2, t_3$ are constant additional costs—so linear costs are modeled as $y = ax + bx + c$.

The code protection cost thus impacts part of the overall cost for a trusted execution. Such an impact is less noticeable when the input/output data protection costs or the execution cost

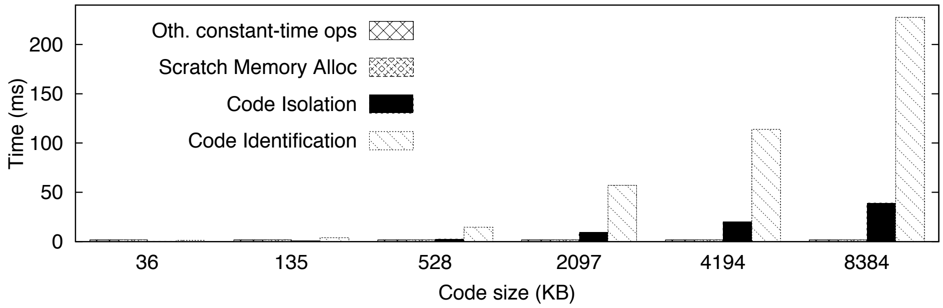


Figure 3.10: Breakdown of the code registration costs inside XMHF-TrustVisor.

outweigh the code protection cost. However, the focus of this chapter is on code identification. Therefore, for the sake of performance modeling, we put emphasis on trusted executions where the code protection cost outweighs the other terms with the following approximation

$$T \approx t_{is}(\mathcal{C}) + t_{id}(\mathcal{C}) + t_1$$

The experimental quantification of these costs in XMHF-TrustVisor is shown in Figure 3.10. We built a set of PALs each having an increasing number of $NOP$ (no operation) instructions. The times for code isolation and identification grow with code size. Other operations, including scratch memory allocation, are code-independent and have constant cost (i.e., $t_1$ overall).

We model the costs of the *fvTE* protocol in a similar way:

$$T_{fvTE} = (t_{is}(\mathcal{E}) + t_{id}(\mathcal{E}) + nt_1) + n\,(t_{is}(in) + t_{id}(in) + t_2) + \\ n\,(t_{is}(out) + t_{id}(out) + t_3) + t_{att} + t_X$$

Here $\mathcal{E}$ is the set of $n$ PALs in an execution flow, and we define $|\mathcal{E}|$ as their aggregated size. Code protection costs are approximated as—notice the per-PAL constant costs ($nt_1$):

$$T_{fvTE} \approx t_{is}(\mathcal{E}) + t_{id}(\mathcal{E}) + nt_1$$

Our protocol is more efficient than the previous approach when protecting the execution flow is less expensive than protecting the whole code base. This can be defined as:

$$\underset{ratio}{efficiency} \quad \frac{T}{T_{fvTE}} \begin{cases} \text{positive, if } > 1 \\ \text{negative, if } \leq 1 \end{cases}$$

A positive efficiency ratio indicates that it is worth having multiple PALs. Instead, a negative efficiency ratio indicates that it is better to protect the whole code base. The (positive) efficiency condition can be defined as follows. First, given the linearity of the code isolation and identifi-

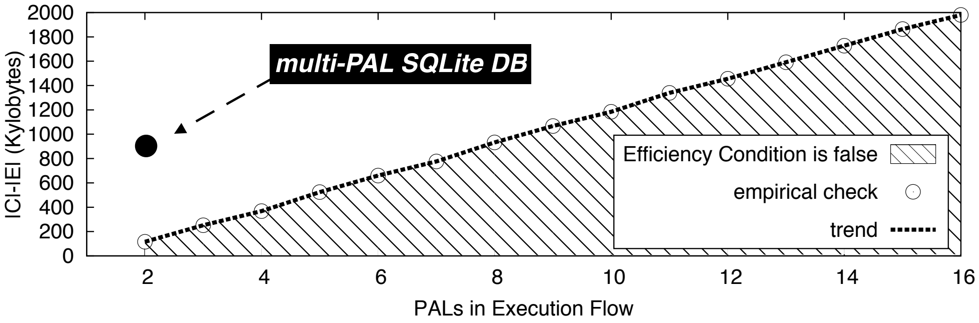


Figure 3.11: Validation of the performance model. The trend line divides the plane in two parts. Points $(x; y)$ = (difference between size of the code base and aggregated size of executed PALs; number of executed PALs) above (resp. below) the line indicate when the presented protocol is more (resp. less) efficient than a monolithic execution. The slope of trend line represents the $t_1/k$ constant in the efficiency condition.

cation costs, we group them as $t_{id}(\mathcal{C}) + t_{is}(\mathcal{C}) = k|\mathcal{C}|$ for some constant $k$, where $|\mathcal{C}|$ is the size of the code base. Then:

$$\frac{k|\mathcal{C}| + t_1}{k|\mathcal{E}| + nt_1} > 1 \quad \rightarrow \quad \frac{|\mathcal{C}| - |\mathcal{E}|}{n-1} > \frac{t_1}{k} \quad \begin{matrix} \textit{efficiency} \\ \textit{condition} \end{matrix}$$

The efficiency ratio depends on both the size of the code base and the size of the execution flow. However, the efficiency condition depends only on their difference (in addition to $n$ and $t_1/k$, the architecture-specific constant discussed later).

We validate the model through an experiment that uses different sets of PALs with cardinality from $2$ to $16$. For each set we varied the aggregated size $|\mathcal{E}|$ (i.e., the size of executed code). We empirically measured the maximum aggregated size for each set for which the *fvTE* protocol is faster than the traditional monolithic approach. This corresponds to the *empirical check* in Figure 3.11. Notice that the trend of these empirical measurements is well approximated by a straight line which divides the plane in two areas: the shaded one where the efficiency condition is false, and the other area where it is true. The slope of the line represents the constant $t_1/k$.

**Discussion.** The constant $t_1/k$ depends strongly on the TCC. In our experiments, it depends on our testbed hardware platform and the software (XMHF-TrustVisor, see Section 3.4.1) that provides trusted computing services. In Flicker [52] both terms are larger due to the interaction with the slow TPM, particularly $k$ for the identification. Instead, technologies such as Intel SGX [189] are expected to reduce significantly both $t_1$ and $k$. However, since the constant also depends on the software that supports trusted executions, it is difficult to predict its trend without running experiments on a real platform.

## 3.6 Other Related Work

**Code identity and trusted executions.** Code identity has been originally defined as the digest of a program's code in [114]. The same definition was later borrowed by several architectures (see Section 2.1). Tools that leverage some of these architectures, such as Flicker [52], TrustVisor [53], Haven [38] do not address the problem of code size inside the trusted environment and execute monolithic applications, whose identity can be verified remotely. In this thesis, we do not change the definition of code identity (i.e., the hash of the binary), and we observe that another way for the client to verify a remote execution is to (be able to) make trust inferences. Therefore, by building a robust chain of trust throughout the modules of a large code base, it is sufficient for the client to verify only part of the chain to infer that the execution of the whole code base was performed correctly.

OASIS [78] proposes to deal with an application whose size is greater than the cache by building a Merkle tree over its code blocks. However, it requires new hardware support, so it does not provide general solution that retrofits existing trusted computing components. Our protocol instead could leverage OASIS by implementing our TCC abstraction (Section 2.3) and, with minimal modifications, it could also include our novel secure storage construction (Section 3.3.4).

The BIND service [73] leverages fine-grained code attestation to secure a distributed system. BIND targets small pieces of code, while our protocol is able to provide execution integrity guarantees of large code bases. Additionally, although small modules could use BIND to build a chain by verifying each other, the resulting construction (similar to that in Section 3.3.1) would not be verification efficient and could incur verification loop issues (Section 3.3.3). Our protocol addresses these drawbacks and guarantees integrity when the client eventually verifies the execution.

Another mechanism [115] proposes an efficient verification primitive in which some code for hash computation is embedded in the software code at compile-time. The program execution is traced by deterministically hashing inputs and outputs of sequences of instructions. Unfortunately, it requires some re-computation, which diminishes the advantages of outsourcing computation. Also, its security relies on the strong assumption that the adversary does not perform code analysis.

One research project related to ours is the On-board Credential (ObC) Project [116, 117, 118]. The ObC Project defines an open architecture based on secure hardware [119, 120] for the installation and execution of credential mechanisms on constrained ObC-ready (typically mobile) devices. It enables a service provider to provision secrets to a family of (installed) credential programs [117], which are executed slice-wise in a secure environment [116], possibly using the TPM's late launch mechanism [118]. Such credential programs are application or platform-specific, while our work is concerned about the efficient verification of executions that are performed on a generic trusted component. The chain of trust among the slices is based on the *slice endorsement token*, containing the family and program-specific secrets, which is created *online* on a per-slice basis. In our case, the chain is explicit in each PAL through a reference to the previous/next PAL's identity, and only needs an *offline* setup (i.e., the process of making the code base available on the cloud provider) performed by the service authors. Also, access to secured data is controlled by (our) construction through the trusted component, allowing secure data exchange among PALs pairwise.

**Defining code modules.** Making modules/partitions out of programs is a programming-language problem that has been widely studied, e.g., in the context of privilege separation [121], parallel program execution [122, 123] and secure distributed computation [69]. Defining such a method for PALs is orthogonal to, and out of the scope of, the presented contribution. We mention however that we built our SQLite-based prototype (Section 3.4.3) by using both static and dynamic program analysis to distinguish the non-active code and remove it, and performing extensive testing to check the correctness of the resulting active code. As an additional example, in another application for secure image filtering, we implemented and protected each filter as a separate task, and then created a secure and efficiently verifiable chain using our protocol, though a different implementation of the TCC primitives (using Flicker [74]).

## 3.7 Summary

This chapter shows that current trends in Trusted Computing create a trade-off between security and efficiency due to the identity assignment for large code bases. To remove such trade-off, we introduce the multi-identity approach through a general protocol that enables efficiently verifiable (at the client) and flexible (at the cloud provider) trusted executions of arbitrarily sized code bases by loading and identifying only the actively executed code. The protocol shows positive results already with a code base of 1MB that implements a real-world database engine. We address the challenge of how to handle efficiently a large state in a trusted execution in Chapter 4.

## Chapter 4

# Support for Large-scale Data in Integrity-protected Virtual Memory

This chapter looks at how we can supply lots of data to a program executed using the execute primitive (recall Section 2.3.1). Two main motivations drive this research step. First, many outsourced services habitually process terabyte-scale data [5, 6, 7, 8, 9, 10, 11]. Second, the state-of-the-art security protocols that are available to provide integrity guarantees for such services make a trade-off between security and functionality.

Elaborating on such trade-off, previous systems either support the processing of small pieces of data [52, 53] or they target specific services such as a database engine [124] or MapReduce applications [39]. Recent work such as Haven [38] has shown how to support unmodified services, and therefore includes large-scale data processing applications. However, as we also mentioned in other chapters, Haven relies on a considerable TCB that includes a library OS. So, on one hand, although a large TCB can give support for the execution of general services, it comes with the downsides of a large TCB as already discussed in Section 1.4.2. On the other hand, limited support for small pieces of code and data and application-specific frameworks severely shrink the domain of services that can be run securely.

These reasons led us to devise **LAST$^{\mathbf{GT}}$**, a system that in spite of its small TCB can handle a **LA**rge **ST**ate on a **G**eneric **T**rusted component. LAST$^{\mathrm{GT}}$ supports a wide range of service applications since it focuses on x86 code, rather than specific frameworks, and only assumes the availability of paged virtual memory—which is widely used in today's processors. In contrast with previous work, LAST$^{\mathrm{GT}}$ is built on a TCC abstraction (Section 2.3.1, enhanced in Section 4.5.5). So it can work on different hardware architectures.

LAST$^{\mathrm{GT}}$ presents three additional key features. First, it focuses on not needing to modify the code of service applications, thereby making their implementation independent from LaSt$^{\text{GT}}$. For example, in our evaluation (Section 4.6) we did not modify the source code of the SQLite database engine. Consequently, LaSt$^{\text{GT}}$ does not require the service developers to know about, or implement anything related to, LaSt$^{\text{GT}}$'s security internals. Second, it imposes no additional verification effort (compared to the protocol in Chapter 3) because it does not heavily change how a client verifies a remote execution. Third, its data structures and procedures can be easily customized. As these mechanisms are implemented at the application-level, they can be easily optimized or upgraded. We defer the overview of LaSt$^{\text{GT}}$to Section 4.2.

### Contributions

- We present LaSt$^{\text{GT}}$'s design, describing how it can protect large-scale data in memory efficiently, and how it enables a client to verify the identity of the service code, data and results.
- We detail how LaSt$^{\text{GT}}$ has been implemented on XMHF-TrustVisor [53] using a commodity platform equipped with a TPM. Also, we discuss a possible implementation using the Intel SGX instruction set and propose some optimizations. In addition, we highlight important differences between the two architectures and how LaSt$^{\text{GT}}$ deals with them.
- We evaluate our XMHF-TrustVisor-based implementation using terabyte-scale data. We show that LaSt$^{\text{GT}}$ has a small TCB compared to state-of-the-art prototypes, and good performance. We also discuss expected improvements of our prototype with an SGX-based implementation.

## 4.1 Previous Work on Trusted Large-Scale Data Processing

We review previous work on trusted executions focusing on how generic systems handle data I/O and how application-specific frameworks handle large-scale data. Also, we review recent work on secure execution based on SGX and additional approaches for ensuring the integrity of computation on large data.

**Trusted Execution Environments for Generic Applications.** Flicker [52], TrustVisor [53], Minibox [62] and Haven [38] all support secure execution (as we mentioned in Section 2.1) and they handle data I/O as follows. Flicker [52] and TrustVisor [53] transfer data to/from the secure environment at execution startup and termination only. The application thus receives all input data upfront. This can be inefficient if not all data is used, and the data size is also bounded by the available physical memory. MiniBox [62] and Haven [38] implement new system calls for dynamic memory and secure file I/O. Both construct a hash tree over the data, encrypt the data and handle I/O through the interface with the untrusted environment for disk access. However, working with a full hash tree in memory does not scale for applications that operate on a large state, since the hash tree itself can consume a large amount of memory. Also, their design exposes several system calls, though fewer than an OS interface, that must be secured against Iago attacks [97].

**Application-Specific Trusted Execution Environments.** $M^2R$ [125] and VC3 [39] were designed for trusted execution of large-scale MapReduce applications. VC3 leverages Intel SGX for guaranteeing integrity and confidentiality of map and reduce functions. $M^2R$ improves the level of privacy by hiding the memory access patterns through a secure data shuffler. Compared with the earlier platforms, these two systems achieve a small TCB at the expense of generality, since they only support MapReduce.

**Other SGX Applications.** Graphene-SGX [56, 222] can run unmodified Linux applications. As it includes a library OS, the same arguments that we used for Haven apply. Scone [40] secures Docker container applications while Panoply [63] secures Linux applications. The former supports multi-threaded container applications and has a larger TCB, mainly due to the `libc` library, while the latter is designed for multi-process applications and has a smaller TCB since it exposes a POSIX-level interface thus leaving the `libc` library outside the enclave. Ryoan [83] secures a distributed sandbox by leveraging SGX to ensure that possibly untrusted code can use, but not leak, sensitive data. These systems [40, 63, 83] are orthogonal to LaSt$^{GT}$ since they focus on secure concurrent/distributed processing and do not target large-scale data. In addition, they expose from tens [40, 83] to hundreds [63] of interface calls, many of which are related to data I/O from/to files. LaSt$^{GT}$ complements them with secure in-memory large-scale data handling that requires no additional interface, but instead relies on page faults and handles data authentication in a scalable fashion.

**Additional Approaches.** TrustedDB [124] and IntegriDB [31] are secure and scalable database systems. Both are application-specific but they have different approaches. TrustedDB uses trusted hardware, as LaSt$^{GT}$ does, and it is fast. IntegriDB only makes cryptographic assumptions and does not leverage trusted hardware, but performance suffers from the use of expensive cryptographic primitives. LaSt$^{GT}$ is not application-specific and does not require expensive cryptographic operations (besides one attestation) as it leverages hardware-based security.

## 4.2 Overview of LaSt$^{GT}$

We give an overview of LaSt$^{GT}$, presenting its key ideas, benefits and challenges.

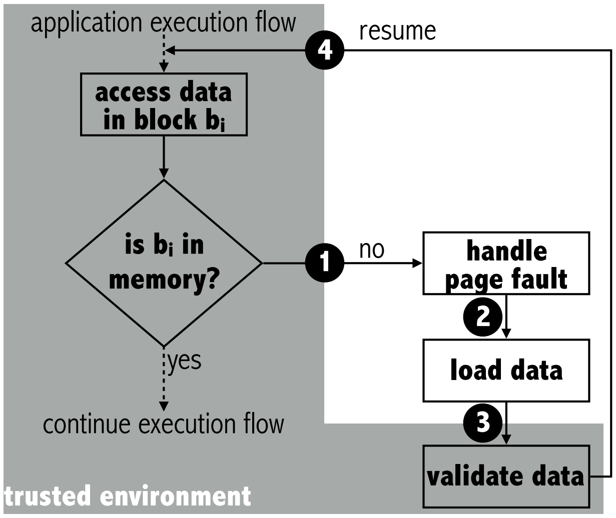


Figure 4.1: Example of a program execution inside a trusted environment that offloads data I/O from storage or network devices to untrusted code.

### 4.2.1 Operation

LaSt$^{GT}$ allows a client to verify the results produced by an application that processes large-scale data on an untrusted platform. The execution of the self-contained service application is secured by means of a trusted hardware component. This component enables the establishment of an isolated execution environment, where the trusted service code is identified, executed and attested. The service processes requests received from a client by reading and writing local state of up to a terabyte of data (in our current implementation) maintained in a set of files. LaSt$^{GT}$ ensures the integrity of the data used during the secure execution, exploiting the key ideas described below. Next, the service generates an attested reply for the client. The reply binds together the identities of the service code, the local state used by the service, the client's request, and the reply. Finally, the client verifies and accepts (if valid) the reply.

### 4.2.2 Key Ideas

The core of LaSt$^{GT}$ is a secure and efficient data loading technique purely based on paged virtual memory and asynchronous handlers (Figure 4.1). LaSt$^{GT}$ presents the large state to the service code as a memory region in its address space, thus allowing it to access the data directly (shaded area, left-side) without requiring explicit calls to privileged code. Data is however not preloaded for efficiency reasons and possible memory constraints. Instead, accesses result in page faults ❶ that LaSt$^{GT}$ handles transparently by moving the data from the untrusted part of the system ❷ into the secure environment. While the service application remains interrupted, the data is cryptographically validated ❸ by a trusted application-level handler (shaded area, right-side) whose execution is asynchronous (i.e., independent from the service application). Only if the loaded data is valid, is the interrupted service allowed to resume ❹.

**Benefits.** Some of the benefits that this design offers include:

- First, it reduces the problem of large-scale data handling to a virtual memory management problem. As virtual memory is *widely* supported, LaSt$^{GT}$ can be implemented on pretty much any Trusted Computing-capable systems, also enabling hardware diversity.
- Second, it keeps the TCB small by not including the OS. Moreover, it does not involve system calls to privileged or untrusted code, that may create vulnerabilities (for instance due to Iago attacks [97]).
- Third, it does not need alterations to the service code since data validation and integrity protection is done transparently.
- Fourth, data validation, integrity protection and (un)loading are handled by customizable application-level code. This allows tuning the authenticated data structures to data access patterns, upgrading deprecated cryptographic algorithms, or devising application-specific data eviction policies.

### 4.2.3 Challenges

Implementing LaSt$^{GT}$ has several challenges:

- *Offloading I/O securely to untrusted code.* Transferring data (e.g., disk I/O) does not represent useful computation for the application, but can increase the TCB size and put peripherals in the trust boundaries. Offloading such operations to untrusted code reduces the TCB but requires means to validate or protect data when it crosses the trust boundaries.
- *Transparency.* Services should only have to access data and perform computation, without dealing with orthogonal issues such as data loading and state management. Performing these tasks transparently simplifies service development and, ultimately, the use of LaSt$^{GT}$.
- *Securely overcoming memory constraints.* Physical memory is limited, especially for secure executions. For example, on a recent Dell Optiplex 7040 [202], SGX is constrained to use only up to 128MB of memory (out of 32GB). Hence, an efficient memory management is needed for handling a terabyte-scale state and the associated authenticated data structure in memory.
- *Dealing with architectural differences.* The platforms that enable trusted executions have dif-

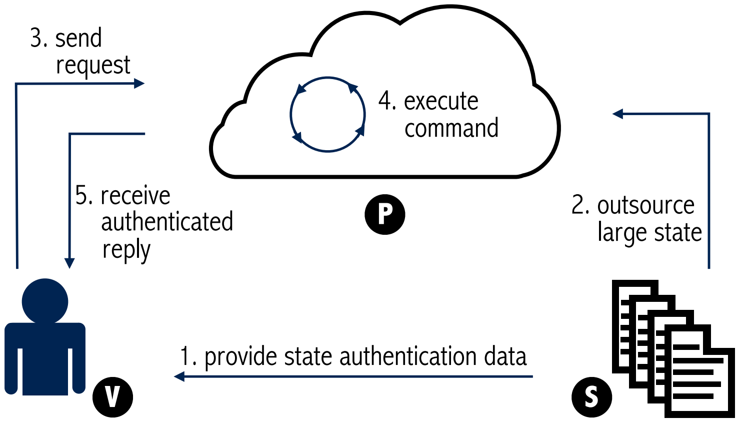

3. send request
4. execute command
5. receive authenticated reply
P
2. outsource large state
1. provide state authentication data
V
S

Figure 4.2: Three-party system model that includes the (client) Verifier, the Provider and the (data) Source.

ferent architectures. This makes hard to hide their differences through abstraction. For example, XMHF-TrustVisor and SGX use completely different mechanisms for secure execution and for paging. So, it is challenging to devise a single design that works for both platforms.

## 4.3 Model

We extend the model introduced in Section 1.3.1 with some additional parties that are specific to LaSt$^{\text{GT}}$.

We assume three parties (Figure 4.2): a trusted source $S$ producing lots of data (*user state*); an untrusted service provider $P$ with significant computational resources; a trusted verifier $V$ that uses $P$'s resources. $P$ and $V$ directly derive from our basic model in Section 1.3.1. $S$ gives authentication data to $V$ (Step 1) and the user state to $P$ (Step 2). $V$ sends requests to $P$ (Step 3), who applies them to the data (Step 4) and returns replies (Step 5) that $V$ checks. Here we focus on a single request/reply exchange with the client. Simple extensions can be devised to deal with subsequent requests and to authenticate updates to the state.

## 4.4 Design of LaSt$^{\text{GT}}$

We introduce the architecture of LaSt$^{\text{GT}}$ (Section 4.4.1) and describe how data is protected at the source (Section 4.4.2), processed by the service provider (Section 4.4.3), and verified by the client (Section 4.4.4). In Section 4.5 we detail implementations on two different Trusted Computing architectures.

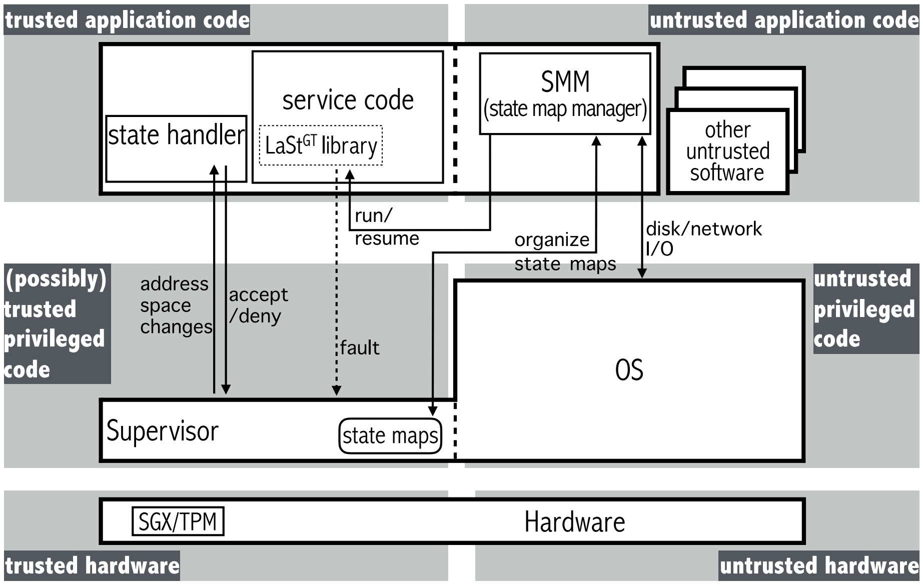


Figure 4.3: LaSt$^{GT}$'s system architecture.

### 4.4.1 Architecture

LaSt$^{GT}$'s architecture is depicted in Figure 4.3. We distinguish between three types of components, namely from the bottom up: hardware (CPU, memory, security chips, disk, Trusted Computing hardware, etc.), privileged code (OS, drivers, etc.) and user-level (or application-level, non-privileged) code. We also distinguish between two types of code execution: trusted code (left-side) and untrusted code (right-side) run inside and outside of the trusted environment, respectively. Depending on the Trusted Computing component, the Supervisor's privileged code must be trusted (e.g., in XMHF-TrustVisor) or can be untrusted (e.g., in SGX), as discussed in Section 4.5. The two hardware/software stacks represent the trusted and the untrusted execution environments. A hardware-based (e.g., a TPM and Intel TXT, Intel SGX, or a secure coprocessor) isolation mechanism prevents untrusted code from tampering with the trusted counterpart.

User-level code includes both untrusted and trusted code. The trusted user-level code is transferred and identified at runtime inside the trusted environment. The untrusted code organizes data and memory for the trusted user-level code, which validates the data before using it. Table 4.1 lists key software modules in LaSt$^{GT}$, where they execute, and what implementation they apply to. In Section 4.4.3 we discuss how they work together to bring data securely into the trusted environment.

| Component | Functionality | Trusted | Trusted Computing arch. | Implementation |
|---|---|---|---|---|
| service code | self-contained general purpose service | ✓ | both | app-level code |
| state handler | check modifications to secure address space; data validation and protection | ✓ | both | app-level code |
| SMM | organization of state in memory; proposal of new memory maps | ✗ | both | app-level code |
| Supervisor | (un)map pages in trusted application address space; switch among components | ✓ | XMHF-TrustVisor | hypervisor |
|  | proposal of modifications to isolated address space | ✗ | SGX | OS-level driver |

Table 4.1: LaSt$^{GT}$'s software components.

### 4.4.2 From User Data to LaSt$^{GT}$-compatible State

The data produced by the data source has to be protected for client verification and structured for processing it securely and efficiently on LaSt$^{GT}$. How data is structured and protected impacts the efficiency of computing the metadata at the data source, the performance of verifying the data at the untrusted provider (e.g., how much data has to be loaded to verify the data that is used) and the verification effort (i.e., time and data required) at the client.

This leads to the following requirements for protecting the data. First, it should be efficient and incremental, so the metadata can be computed as data is produced. Second, it should enable piece-wise data validation, so to handle only subsets of data in memory. Third, it should enable constant time verification by the client.

LaSt$^{GT}$ provides these features by pre-processing the user data into a *state hierarchy* of sub-components, consisting of blocks of user data at the leaves and structural and authentication metadata higher up in the tree (Figure 4.4). Each component has a cryptographic identity that depends on its sub-components' identities, and ultimately on the user data. These identities form a cryptographic hash tree optimized for dealing with a large state, as described in Section 4.5.2.1. The structure is built using an incremental procedure and allows piece-wise data loading and validation through the concepts of data *chunk* and *block*. Also, it enables constant time verification at the client by checking the state identity, i.e., the identity of the root.

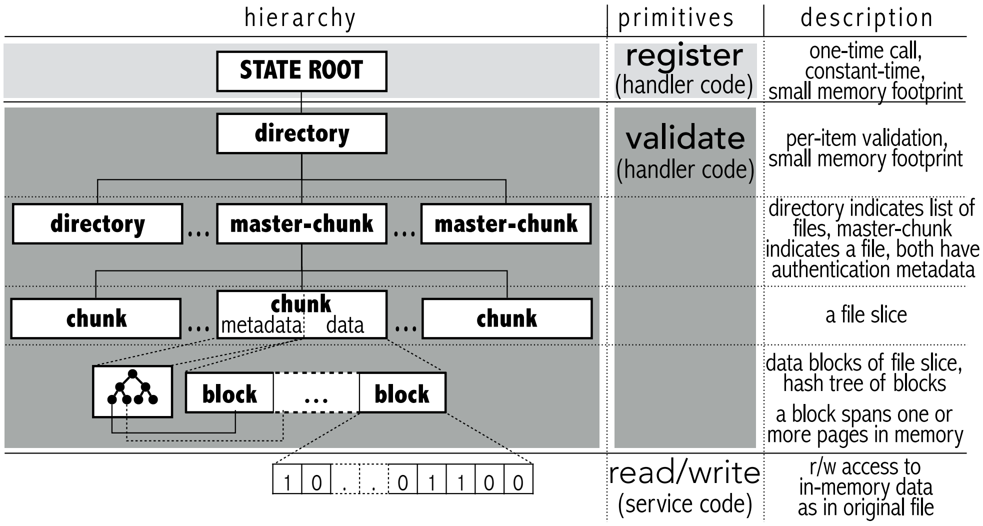


Figure 4.4: State hierarchy. A directory can contain several master chunks and (sub-)directories. The relevant primitives (register, validate) build a chain of trust between the service reads/writes and the state root verified by a client.

### 4.4.3 Data Processing at the Untrusted Provider

We describe how LaSt$^{GT}$ manages the service code execution (Section 4.4.3.1), how data is read from disk (Section 4.4.3.2), loaded into the trusted environment (Section 4.4.3.3) and reclaimed (Section 4.4.3.4).

#### 4.4.3.1 Service Execution

A LaSt$^{GT}$ execution begins by registering the state root identity (provided by the data source) with the state handler in the trusted execution environment (Section 4.5.2.3). This is a one-time procedure that must be secure since the integrity of the entire state hierarchy, including the user data, depends on the correctness of the state root identity. It is not necessary to load the full state upfront, since the root is sufficient to validate any data that is loaded during the execution. After the execution terminates, the registered (root) identity of the input data is also included in the attestation so that a client can verify it.

The service code is then executed and it uses regular I/O calls to access user data, though without issuing any system call. I/O calls use the LaSt$^{GT}$ library to access the user data. The library has a memory-mapped view of the state hierarchy, which it traverses beginning from the registered state root to access the data. As the data, including the metadata, is not available upfront in the isolated memory, the library execution is interrupted by page faults, which are handled by the Supervisor, as described in the next section.

**In-Memory Embedded Locators (IMELs).** A naive loading would allocate (for example) $2^{40}$ virtual addresses upfront for a state hierarchy of 1TB, and each page access would trigger a page fault. This is feasible on 64-bit architectures, but the SMM would have to ask the OS for many virtual address mappings, most of which may not be used. Also, the OS may be required to remap physical pages and do some paging to disk. This would occur similarly for the Supervisor while managing pages in the trusted execution environment, thus adding overhead. In addition, some platforms, e.g., XMHF-TrustVisor, use 32-bit addresses and so have architectural limitations.

To deal with these overheads and constraints, LaSt$^{GT}$ uses IMELs to reuse addresses and memory. IMELs are memory pages embedded in the state hierarchy at runtime between a parent and a child component (e.g., a master chunk and a chunk). Specifically, a parent component points to an IMEL that contains the address of its child component, rather than pointing to its child component directly. By not loading the IMEL in isolated memory when the parent component is first loaded, this makes the service code raise a page fault on a memory page that contains an address, rather than the child component. So IMELs just provide positions in memory. They can be filled at runtime and loaded in isolated memory together with the child components they reference. Similarly, they can be unloaded together with the component, so to reuse the allocated memory with other data.

#### 4.4.3.2 Loading state from disk into untrusted memory

The Supervisor delegates to the untrusted SMM the handling of page faults related to data that is not yet in main memory. In particular, it does so by transferring control and providing the fault address to the SMM (Section 4.5.2.5). Performing such tasks at user level moves the code out of the Supervisor's TCB, which is important in architectures where the Supervisor must be trusted. The SMM uses the fault address to figure out what state component (see hierarchy Figure 4.4) should be loaded from disk. It then loads the component from disk and places it into untrusted memory. For any component that is in memory (e.g., chunks, IMEL, etc.), the SMM maintains a map item in a map list (Figure 4.7, Section 4.5.2.2). The map list is updated before the SMM returns control to the Supervisor. These maps will be used by the Supervisor for moving pages into the trusted environment, and by the state handler for validation.

#### 4.4.3.3 Authenticated lazy loading from untrusted memory

LaSt$^{GT}$ optimizes loading data from untrusted memory into the isolated memory using authenticated lazy loading, i.e., pages or blocks are loaded on demand (Section 4.5.2.4). This is done either after a page fault on data already in untrusted memory or after data has been fetched from disk. In particular, the Supervisor handles page faults by using the memory maps to locate the pages; if the Supervisor is trusted, it maps the pages in isolated memory; while if it is untrusted, it provides memory pages in isolated memory where the data will be copied into. The state handler is then invoked to validate them.

Page and data validation are performed within the trusted environment by the state handler before the service code can access the data. This ensures that the library (and thus the service) can only access valid data—there is no data validation performed within the service code, so the operation is fully transparent to the service code. The procedure is performed at the user-level because the Supervisor may be untrusted depending on the Trusted Computing-architecture.

#### 4.4.3.4 Reclaiming memory

The untrusted SMM can reclaim state components from isolated memory by updating the map list (Section 4.5.2.5). The reclaim is validated and accepted (or denied) by the state handler and, only when it is accepted, the Supervisor is allowed to withdraw the reclaimed pages from the trusted execution environment.

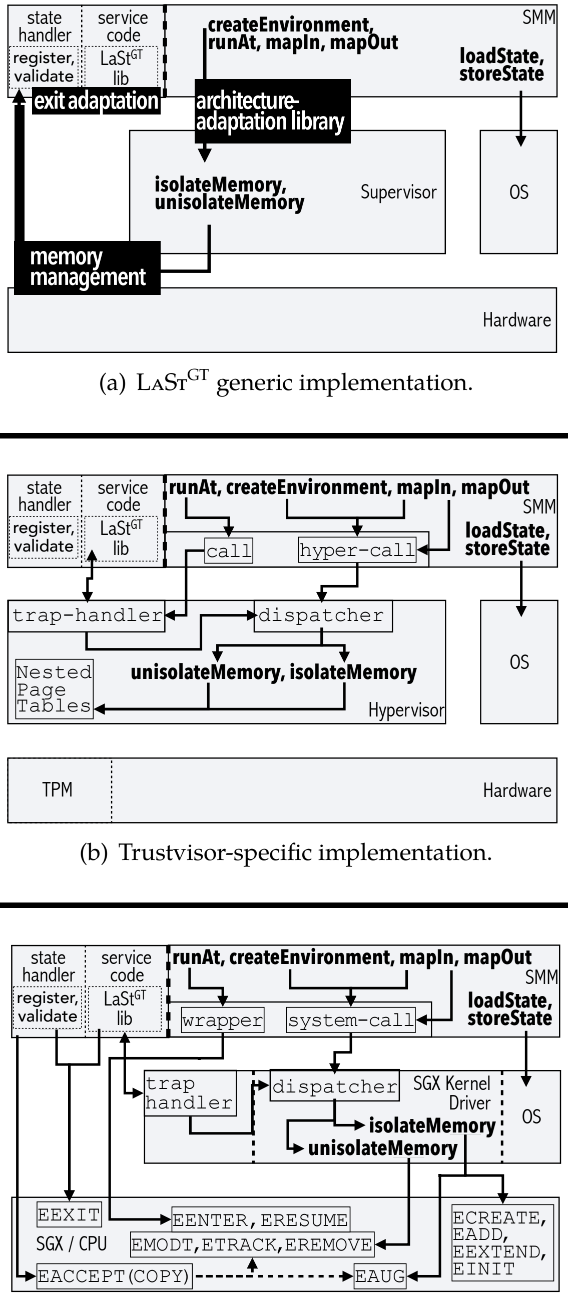
### 4.4.4 Client Verification of a Remote Execution

As in Chapter 3, the verification of a remote execution is equivalent to verifying a hardware-based execution attestation. The attestation vouches for the identities of: 1) the executed code, 2) the input and 3) output data, 4) a client-provided nonce (Section 4.5.2.6). A successful verification validates the signature, using a manufacturer-certified public key (or an Attestation Verification Service [194]), and makes sure that the attested identities are the intended ones.

## 4.5 Implementation of LaSt$^{\text{GT}}$

### 4.5.1 Overview

Figure 4.5 zooms into the architecture (Figure 4.3), detailing the implementation of LaSt$^{\text{GT}}$. In particular, Figure 4.5(a) abstracts the specific details of the hypervisor-based implementation (Figure 4.5(b)) and of the SGX-based implementation (Figure 4.5(c)). This helps identifying common parts of LaSt$^{\text{GT}}$ whose code can be shared across different Trusted Computing-architectures.



(a) LaSt$^{GT}$ generic implementation.

(b) Trustvisor-specific implementation.

(c) SGX-specific implementation.

Figure 4.5: LaSt$^{GT}$ abstraction of non-common mechanisms (Figure 4.5(a), black boxes). Trusted Computing-architecture-specific implementations (Figure 4.5(b) and Figure 4.5(c)).

The primitives in bold are common across implementations. Inside the SMM, they allow the untrusted user-level code to set up the environment (**createEnvironment**) for the execution of the trusted user-level code, to run it (**runAt**), to map state components data and metadata in and out (**mapIn, mapOut**) of the isolated address space—though the state handler validates them first—and to manage state components on disk (**loadState, storeState**) through the untrusted OS. At the privileged level inside the Supervisor, the primitives (**isolateMemory, unisolateMemory**) allow the Supervisor to provide memory pages to (or to withdraw them from) the isolated address space of the trusted application code according to the maps configured by the SMM. The attestation primitive that is used by the state handler is not shown to simplify the description and the figures.

Several primitives need to interact with hardware and software that are specific to the Trusted Computing architecture that is used (abstracted by the black boxes in Figure 4.5(a)). First, the architecture-adaptation library includes code to execute a hyper-call or a system-call handled by a dispatcher, or to execute an instruction wrapper, or a call that traps into a trap-handler. The memory management and the exit adaptation boxes have very specific functions. The former touches the state handler due to the `EACCEPT` and `EACCEPTCOPY` SGX instructions (Section 4.5.4) that the state handler runs to accept changes to enclave pages. The latter instead hides the `EEXIT` instruction in SGX, or a simple return of a function in XMHF-TrustVisor, to terminate the execution. Finally, LaSt$^{\text{GT}}$ simply uses the OS and standard libraries for managing state on disk.

Inside the trusted application code, the LaSt$^{\text{GT}}$ library mainly navigates the state hierarchy, while the register (Section 4.5.2.3) and validate (Section 4.5.2.4) primitives hide the details of the memory maps and of the authenticated data structure embedded in the LaSt$^{\text{GT}}$-compatible state. Also, in XMHF-TrustVisor, the trusted application code just directly calls the hypervisor for the attestation (not shown). In SGX, instead, it has to run dedicated instructions to terminate, to accept pages and to attest, thereby making the code more dependent on the Trusted Computing architecture.

To summarize, LaSt$^{\text{GT}}$ can deal with the architectural differences between XMHF-TrustVisor and SGX through small adaptations within a single design. Next, we describe the architecture-independent components of LaSt$^{\text{GT}}$ (Section 4.5.2), our implementation for XMHF-TrustVisor (Section 4.5.3), and a design for SGX (Section 4.5.4) in more detail.

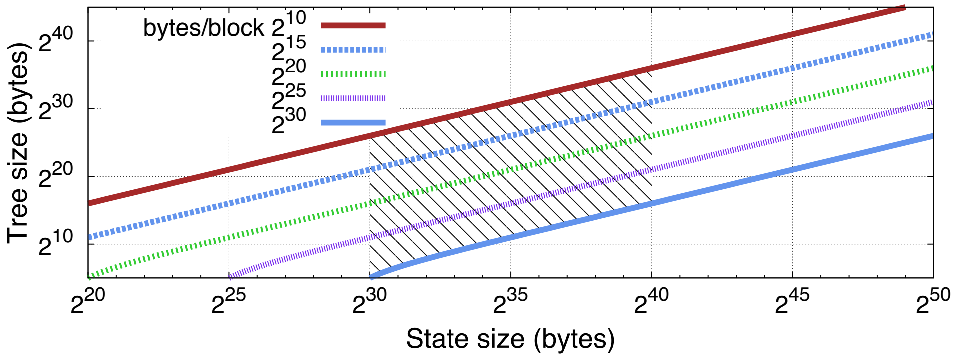


Figure 4.6: Hash tree size (y) as a function of the state size (x) for different block sizes, with a 32 bytes hash (e.g., SHA256). Shaded area is our target.

### 4.5.2 Trusted Computing-architecture-independent Details

#### 4.5.2.1 Building the state

As discussed in Section 4.4.2, the user state and its meta data is organized in a state hierarchy (Figure 4.4). The *state root* contains one hash value that both the service and client code rely on to validate the authenticity of the data. A *directory* is a set of (sub-)directories and master chunks. A *master chunk* maps one-to-one to a file in the filesystem. Each master chunk includes a set of chunks, each of which corresponds to a contiguous sequence of user data in the file. The user data in a chunk is further logically divided into a set of *blocks*. Each chunk also includes metadata, called a chunk descriptor. It contains a static hash tree built from the chunk's data. The leaves of the tree are computed by hashing contiguous bytes of a block in the chunk. The root of the tree represents the identity of the chunk. The identities of the chunks are hashed to form the identity of their parent master chunk, and so on up to the root, which is the identity of the entire state.

Only a configuration file and two parameters (chunk and block size) are required to build a LaSt$^{GT}$-compatible state. This information is defined by the user. The configuration file is a list of files (each one producing a master chunk) and directories to be included in the state.

This design of the state hierarchy allows to manage data in memory very efficiently. We can load fixed-size chunks from disk as needed, without dealing with the entire state data at once. Then we can load blocks from untrusted memory into the trusted execution environment, possibly batching the transfer of the pages spanned by a block. Also, as the authentication metadata is distributed across the state hierarchy, we can easily and locally validate data blocks in a chunk and update the hash tree when a block is modified. In fact, the hash trees of other chunks can remain on disk.

Distributing the authentication data is important for our target state sizes. A single file-wide (as in Minibox) or disk-wide (as in Haven) hash tree has several drawbacks in comparison. First, a single tree can take up to gigabytes (Figure 4.6, top-right of shaded area). Second, the size of the tree adds complexity to cache it in secure memory and in untrusted memory. Finally, one could opt to load a data block together with a short membership proof (linear in the height $h$ of the tree). However, when using a single tree for a 1TB of user data ($2^{40}$ bytes) and small blocks ($2^{10}$ bytes), the hash tree is tall $h = 31$, so the proof is large, i.e., $(h-1)\ nodes \times 32\ bytes/node = 960\ bytes$ or $93\%$ of block size, and verifying it takes many (i.e., $h-1$) hashes.

#### 4.5.2.2 Maps for State Organization and Memory Management

LaSt$^{\mathrm{GT}}$ uses memory maps for state and memory management. Figure 4.7 shows some entries in an example map list that the SMM uses to store type, address and size (in pages) of the memory the entry represents. The SMM uses different map types for metadata and data (currently 15 types, including those for IMELs, debugging and performance measurements), including a special one to reclaim a map. The maps are also shared with the Supervisor which manages the physical pages (in SGX, or the memory access permissions in XMHF-TrustVisor), and with the state handler that validates changes to the address space.

Memory accesses to data that is not in the trusted environment trigger a page fault and the Supervisor is invoked. The Supervisor looks up the page fault address into the memory maps. If the address points to data mapped in memory, this is a *map hit* and the Supervisor performs lazy loading in the secure environment (Section 4.5.2.4). If the address instead points to non-mapped data, this is a *map miss* and the Supervisor triggers the procedure for loading state from disk or for shutting down the execution in the case of an illegal access (Section 4.5.2.5).

The state handler uses the maps to locate metadata for validation and to ensure that pointers to supposedly non-mapped components do not incorrectly dereference mapped components—they must produce a fault. Notice that the initial maps (if any) must be embedded in the trusted user-level code and so included in the code identity eventually verified by a client. So, initial and subsequent maps can be trusted. To avoid tampering from the SMM, who might maliciously swap map types (e.g., the state root map type for a IMEL type) for instance, the state handler maintains a secure copy of the map list.

| map list | | |
|---|---|---|
| type | address | pages (=length) |
| D | 0x0b000000 | $2^{18}$ (=1GB) |
| H | 0x4b000000 | 1 (=4KB) |
| C | 0x4b001000 | $2^{15}$ (=128MB) |
| | ... | |

Figure 4.7: Example of a map list. Items describe type (D=dynamic memory, not included in state hierarchy; H=in-memory embedded locator; C=chunk), location and size of a map. Other types (and entries) are available for: root, directories, master chunks, chunk metadata, input and output maps.

#### 4.5.2.3 State Registration

State registration is the first code executed in the trusted environment. It allows the state handler to receive the state root map. The map contains a single hash value for authenticating the metadata and data in the state hierarchy. The state handler uses the register primitive to copy the root hash to a static variable representing the input state—once set, it cannot be overwritten nor reset (without beginning a new trusted execution).

#### 4.5.2.4 Normal Execution and Lazy Loading

The service execution is triggered by the LaSt$^{GT}$ library that begins the execution in the trusted environment. The library initializes the state base pointer to the state root map set up at registration-time. Then, starting from the state root, it walks the state hierarchy by following the pointers between parent and child components.

**Correct access of a state component.** LaSt$^{GT}$ has to ensure that the library: (i) produces a page fault when it accesses a non-validated state component; (ii) will find valid content in memory and hence (iii) can be resumed only in this case. The state handler ensures (i) by peeking into a state component being loaded to check that it has a pointer to an IMEL page that is not yet mapped in the isolated memory (e.g., $csc_3$ in Figure 4.8). The handler ensures (ii) by cryptographically validating the state component using the validate primitive. For example, in the case of a chunk, the primitive checks the chunk's hash tree and whether its root matches that stored in the parent master chunk. For an IMEL, instead, the state handler simply checks that it contains the address of valid state component being loaded jointly. How we can ensure (iii) depends on the used Trusted Computing architecture, so we defer the explanation to Section 4.5.3 and Section 4.5.4.

**Loading metadata vs. Loading data.** Except for the original user data in a chunk, the rest of the state hierarchy is considered metadata. Metadata (directories, master chunks, chunk metadata)

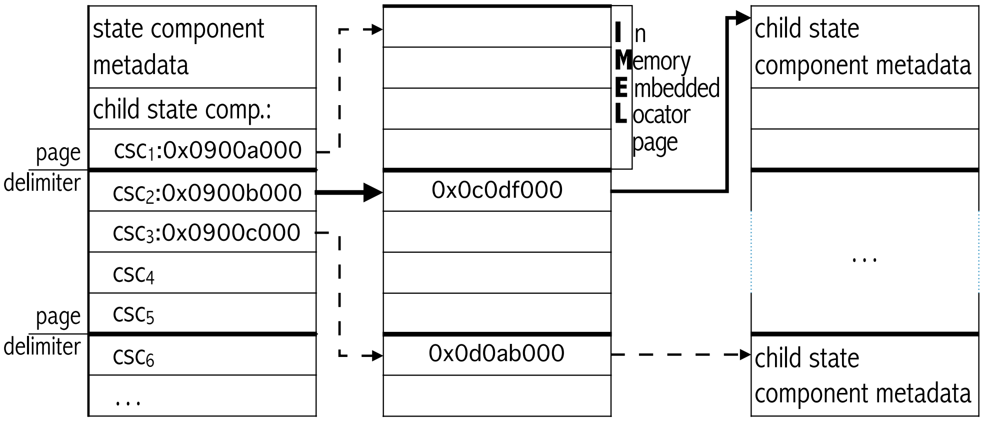


Figure 4.8: IMEL pages contain positions of data in memory. They allow the LaSt$^{\text{GT}}$ library to access a child state component (e.g., a chunk, $csc_2$), and they help the untrusted code to locate such component and load it. Other IMELs (e.g., at $csc_1, csc_3$) that are not accessed are not loaded.

and data have different types of maps. This allows the Supervisor to behave differently when the library produces a page fault while walking through the state components. If the fault address is map-hit by a data-typed map (only containing data), one data block is loaded. In this case, the state handler just performs cryptographic validation. If the address is instead map-hit by a metadata-typed map (only containing metadata), then the entire map is loaded. The rationale is that metadata is small (compared to data, see evaluation Section 4.6) and can best be validated immediately. This later allows validation of a data block in constant time—it is sufficient to check if a block's hash matches the associated hash tree leaf.

#### 4.5.2.5 Loading Data From Disk and Reclaiming Maps

If the Supervisor cannot find a map that covers the page fault address, then that is a map miss. A map miss only occurs on IMEL pages unless bugs result in illegal map misses. The Supervisor transfers control to the SMM to handle the map miss. The SMM uses the fault address to locate the IMEL and needs metadata in untrusted memory to locate the data on disk. For this reason the SMM maintains shadow copies of parent state components—treated later (Section 4.5.3, Section 4.5.4) due to differences in the trusted address space configuration between the considered Trusted Computing architectures. So the child state component is loaded from disk into an arbitrary free memory range and the associated IMEL page is updated. Then the SMM creates map entries for both the IMEL page and the child component and informs the Supervisor of the new maps. The Supervisor retries handling the fault whose address should result now in a map hit.

When a map miss does not reference a IMEL page (e.g., the null $0 \times 0$ address), this is considered a software bug. The SMM thus triggers an execution shutdown (segmentation fault).

**Reclaiming maps.** The SMM reclaims a map $m = (type, address, pages)$ by inserting in the list another entry $m' = (\textsc{reclaim}, address, pages)$ with a special reclaim-type. Finally, the handler checks the validity of the reclaim (for instance, as explained next, that the reclaimed component has no modified descendant component) and accepts giving the map back to the SMM.

**Reclaiming maps in the presence of modified data.** While there is a modified state component mapped in (e.g., a chunk), the state handler never accepts the reclaim of any map of that component's ancestors (e.g., a master chunk). The hash tree in fact may not be up to date with the modifications. If a reclaimed component has no child components in the secure environment, the state handler updates the hash tree (if necessary) and accepts the reclaim. At attestation-time, the state handler similarly updates the hash tree root. This allows clients to know the identity, and verify the integrity of, the output state.

#### 4.5.2.6 Attestation and Remote Verification

LaSt$^{\text{GT}}$ combines in the attestation a client-provided nonce and the identities of the registered state, the output state (if any), the client request, the reply, the trusted application code. The attestation is performed as in Chapter 3 using the attest primitive. The attestation parameter (i.e., the attested data) is however enriched by the trusted state handler's code by combining the identities of the registered/output state in addition to the identities of the request and the reply.

The state handler initially receives from the SMM specially-typed maps that contain the request, the reply and the nonce. It accepts the maps and, respectively, it hashes the request map to save the request identity, it saves the nonce, it zeroes the reply map that is later filled by the service code. After the service code execution terminates, when the state handler runs again to perform the attestation, the data in the reply map is also hashed to get the identity. The state handler then hashes the identities of the state, the request and the reply together with the nonce. The result can be either hash-chained to the trusted application code identity and attested in XMHF-TrustVisor, or provided in input for the cryptographic report that includes the trusted application code identity, in SGX.

Assuming that the client receives the reply (and recalling our model, see Section 4.3), the client knows all of the attestation parameters and can therefore verify the attestation. In particular, the client establishes trust in the reply only if the identities that are combined in the attestation match the expected ones.

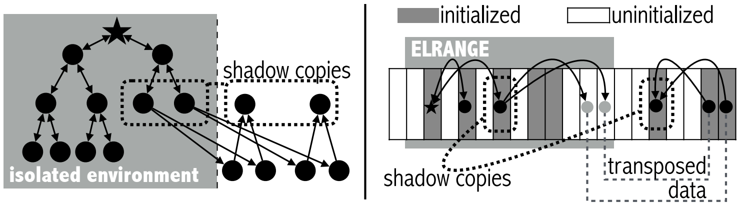


Figure 4.9: In the trusted environment, parent state components can reference child ones in untrusted main memory either *directly* using their addresses (in XMHF-TrustVisor, left-side), or *indirectly* through their transposed addresses in uninitialized pages within the enclave's secure range, where they will be loaded (in SGX, right-side). Referenced components, that are not yet in untrusted memory, are located and loaded using metadata shadow copies.

### 4.5.3 Implementation in XMHF-TrustVisor

A representation of the XMHF-TrustVisor-based implementation is displayed in Figure 4.5(b). For background about XMHF-TrustVisor, we refer the reader to Section 2.2.1.

The architecture of XMHF-TrustVisor simplifies (compared to SGX) the implementation of LaSt$^{GT}$ in two ways. First, the hypervisor can keep the service interrupted (after a page fault) until successful validation by the state handler, so the service always accesses valid data when it is resumed. Second, the hypervisor can modify the trusted application code's page tables, so it can be trusted to load (and similarly unload) data at the correct position in the trusted execution environment.

The extended hypervisor orchestrates a LaSt$^{GT}$ execution based on the feedback received from the application-level, which returns true or false as the result of data validation. The hypervisor begins by running the state handler supplying the maps of the state root, the heap memory, the request, the reply and the nonce. Feedback from the state handler is a return value: $0$, registration (or validation) unsuccessful; $1$, registration (or validation) successful. If successful, the service code can be executed until termination, or until a page fault occurs by accessing a non-isolated state component as shown in Figure 4.9 (left-side). On termination, the hypervisor asks the state handler to protect the integrity of modified pages in the state hierarchy up to the root and to attest the result.

On page fault, the hypervisor checks the maps. In the map-hit case, it provides the data and the page fault address to the state handler for validation and, if successful, it resumes the service code. In the map-miss case, the hypervisor provides the page fault address to the SMM. The SMM uses it to locate the metadata shadow copy of the (isolated) state component, which is used to load the missing (child) state component in memory. The SMM then returns the new maps to the hypervisor.

The hypervisor switches control flow between the service code, the state handler and the SMM by alternating the execution of the VM with the untrusted OS and of the VM with the service code. Namely, it updates the instruction pointer to one of the entry points (service code, state handler, SMM) or to the faulted service code instruction. In the case of an entry point, the hypervisor pushes a special instruction pointer on the stack so the code traps back into the hypervisor when it returns.

The hypervisor isolates a map by ensuring that: its pages are (lazily, Section 4.5.2.4) inserted in the page tables of the trusted application code; the nested page tables are configured to grant trusted application code access to the associated physical pages, while denying the SMM and the OS access to them. Un-isolating a map works in the opposite way.

### 4.5.4 On the feasibility of LaSt$^{GT}$ Using Intel SGX

A representation of LaSt$^{GT}$ implementation in SGX is displayed in Figure 4.5(c). For a background about Intel SGX, we refer the reader to Section 2.2.2.

#### 4.5.4.1 Main Implementation Challenges and Solutions

In SGX, the memory management and the secure control flow management are slightly more complex (compared with XMHF-TrustVisor) for three reasons. First, addresses and content of enclave memory pages to be added or removed at runtime must be checked and accepted by the enclave at the application-level, without help from trusted privileged code. Second, enclave code can access untrusted memory outside ELRANGE. While this is useful to load (and then validate) data from untrusted memory, it opens the risk of using incorrect data inadvertently. Third, untrusted code can run/resume enclave code at any time. Hence, concurrency issues may arise within the enclave, particularly when resolving a page fault or performing an attestation.

LaSt$^{GT}$ deals with the first challenge as follows. Since untrusted code cannot start the enclave execution at an arbitrary instruction, we build the enclave with separate entry points for the service code and the state handler. This allows running the state handler while the service code is interrupted.

A mechanism is required to ensure that the service code can only be resumed, and not re-executed, after an interruption. As untrusted code can behave arbitrarily, it can restart the enclave at the service code entry point. We thus build the enclave with a single area (SSA) per entry point (TCS) to save the processor state on interruption (e.g., due to a page fault). As an interruption consumes one such area and the CPU requires one SSA to be available to start the enclave, this prevents multiple executions at the service code entry point, before the service code terminates.

When the state handler is executed, the handler has to validate the position where memory pages are supplied, and the content these pages should have. The position is the address of the page where the fault occurred during the service code execution. Such address is found using the CR2 control register where the CPU stores the fault address. However, reading CR2 requires privileges, so the application-level enclave code cannot do it. Also, the state handler cannot trust the (untrusted) SGX driver to supply it correctly, although it expects the driver to supply the memory pages. Fortunately, the CPU includes the value of CR2 in the enclave's secure region when the execution is interrupted. So the state handler has access to a trusted address and can check that a map (in the list) covers it.

LaSt$^{GT}$ deals with the second challenge as follows. Ensuring that the service code does not access incorrect data, particularly in untrusted memory, requires validating data and pointers to data. Validation must occur while the data is inside the secure region. Pointers to data must have addresses inside the secure region. Also, if the data has not yet been accessed, referenced pages must be unavailable so to produce a page fault.

Validating the content of the memory pages is tricky because they are not available, otherwise the service code could be resumed. The challenge is to enable the resumption of the service code only after the pages are available with the right content. This is solved by validating the content elsewhere and leveraging SGX to fill the pages appropriately as follows. At creation-time, we include in the enclave a buffer large enough (e.g., 4MB) to contain a state component metadata or a data block. We program the state handler to copy the data from non-enclave memory to the internal buffer and to validate it. Besides validating the integrity of the data, the state handler also checks that any address referencing an IMEL or a child state component falls within the secure region. This prevents the service code from accessing untrusted memory. We discuss later where the data is placed in untrusted memory.

Assuming the data is valid, the next step is placing it correctly so that the service code can be resumed and access it. We resort to the EACCEPTCOPY SGX instruction to do it. The instruction allows to copy an available enclave page into an uninitialized enclave page and to initialize it. So the state handler executes the instruction to copy a page of the buffer containing the data into a still unavailable page that the interrupted service code cannot access. After this step, the service code can make progress since the page is available and contains valid data. The procedure can be easily extended to batch the validation and acceptance of a set of pages. We mention that SGX provides another instruction for accepting memory pages, i.e., EACCEPT. This is useful for dynamic memory allocations (e.g., our dynamic memory map), that are supposed not to contain sensitive data initially—in fact they are zeroed. However, using it in the previous step would not be secure, because it initializes the memory pages (which become accessible) before they are filled with valid data.

Now we explain how our maps are used as the data has to be transferred from untrusted memory to the trusted buffer and then copied elsewhere in the enclave's memory. In XMHF-TrustVisor, since we can (un)isolate a single memory page, one map per component is sufficient. In SGX, instead, the SMM cannot load data directly into the enclave region, and the state handler should have some means to locate data in untrusted memory. Our solution is mapping each state component into two map lists, $M_1$ and $M_2$, thereby having two maps. $M_1$ follows our original description: it expresses where memory and data are or should be placed within the enclave region—so the addresses belong to the enclave region. $M_2$ is logically derived from $M_1$ by transposing the address of each map into an address in untrusted memory. So, the SMM uses $M_2$ to arrange in untrusted memory the state components that are loaded from disk, and it uses $M_1$ to arrange the same components in trusted memory. Instead, the state handler uses $M_2$ to (un)load data to/from untrusted memory, and uses a private copy of $M_1$ to ensure the correct position of the maps in the enclave's secure region.

LaSt$^{\text{GT}}$ deals with the third challenge as follows. As the attestation is performed by the state handler, the handler has to make sure that the service has indeed terminated and will not modify the state (e.g., if it is re-executed) during the attestation. We address this concurrency problem by synchronizing the service code and the state handler using shared variables transparently inside our linked library. Notice that in a multi-core environment, such shared variables can be managed in transactional regions with Intel TSX (Transactional Synchronization Extensions), which is available on the Skylake microarchitecture and compatible with SGX [189, 6.14].

#### 4.5.4.2 Proposed SGX Optimizations

We propose two optimizations for SGX, both related to data I/O. The first one aims to reduce the number of page writes for mapping data inside the enclave's memory. We propose two new instructions, `EAUG_DIRTY` and `EACCEPT_HASHCOND` to achieve this objective. The second one aims to reduce the number of execution transitions between enclave code and untrusted code for the page trimming procedure, which is a two-phase protocol as in [189, 3.5.9]. We propose another instruction, `EPRE_ACCEPT` to achieve this second objective.

**Problem and proposed optimization for mapping data in.** Loading 1 page (4KB) into an initialized enclave, within ELRANGE, implies writing $4$ pages (16KB): (1) when the Supervisor runs EAUG that zeroes an uninitialized page $p_1$; (2) when the SMM loads $4$KB of data from disk into an untrusted page $p_2$; (3) when the enclave copies $p_2$ into an in-enclave buffer page $p_3$ for validation; (4) when the enclave runs EACCEPTCOPY that copies $p_3$ into $p_1$ and initializes $p_1$ for the service to access it.

We claim that $4$ page writes are optimal with the current SGX instructions [189] though they have high overhead. So we propose two new instructions to reduce writes to $2$.

1. The Supervisor executes EAUG_DIRTY to copy a source dirty (i.e., containing data) page from untrusted memory to an in-enclave uninitialized (i.e., marked as pending) page.
2. The enclave executes EACCEPT_HASHCOND that hashes the uninitialized page and compares the result with an in-enclave-memory hash value. If they match, the page is initialized for enclave use, otherwise a violation is reported.

The first step includes $2$ page writes: initially, the OS writes the data in an untrusted memory page; then EAUG_DIRTY writes an in-enclave page. The second copy is similar to EADD's page copy (Section 2.2.2), except that EADD only works at enclave-creation time, while EAUG_DIRTY (like EAUG) only works at runtime. Also, the page hash can be computed with SHA-256 CPU instructions, already used by SGX. So we believe these functions can be realized by extending the SGX microcode.

**Problem and proposed optimization for trimming pages.** Trimming pages requires a two-phase protocol between the Supervisor and the state handler. The cost can be amortized by trimming pages in batch (e.g., large maps with thousands of pages). However, trimming few pages (e.g., small metadata maps) still requires entering/exiting the enclave twice.

We propose an EPRE_ACCEPT instruction to pre-accept a future trim request and merge the two phases. Roughly—the notation $SGX.y$ denotes an operation that requires the execution of an SGX instruction, while other text above the arrows simply refers to protocol messages:

$$Supervisor \xrightarrow{prepare-trim} Handler \xrightarrow{ok,\ SGX.pre-accept} Supervisor \xrightarrow{SGX.trim} Supervisor$$

This requires entering/exiting the enclave just once. We believe this optimization is also feasible with a microcode extension.

### 4.5.5 How the TCC primitives are extended

We have to extend the primitives in Section 2.3.1 in order to support LaSt$^{\mathrm{GT}}$'s virtual memory management. Since the management is performed by suitably handling interruptions of the secured service application while it is executing, support for LaSt$^{GT}$ can be fully implemented within the execute primitive. In addition, small changes are required to the verify primitive. This clarifies how we provide such a support "below" the primitives (Section 2.3.1).

**An enhanced** execute **primitive.** Algorithm 23 and Algorithm 24 (dark background, bottom-side) highlight how the execute primitive changes. We point out that, in XMHF-TrustVisor, in-memory data management is performed at the level of the hypervisor (and thus not shown) which provides memory and data to the trusted execution. In SGX instead, data management is deputed to the untrusted application-level (AEX code block, lines 14-24).

The AEX block is executed when the enclave is interrupted, for instance due to a page fault. In the case of a map miss (i.e., the needed data is not in memory), the SMM is called to load the data in memory (line 16); we display such direct call for clarity, however, the SMM can be implemented outside the primitive and then used as we describe in a note below. Once the data is in memory, the AEX code block leverages the OS driver to run privileged instructions to supply memory to the enclave (line 18, though the instructions are not detailed for brevity). For security reasons however, the enclave code must accept any supplied memory before using it, and also has to retrieve and validate data from untrusted memory. These operations are performed by letting the enclave code start at a different entry point (line 20). If validation is successful, then the original execution flow can be resumed (line 21). If the interruption is unexpected, or the untrusted application-level is unable to resolve the page fault, then the execution is simply terminated (line 23), so no results or attestations are produced.

The reader can easily check that the implementation is compatible with the primitives introduced in Section 4.5.1. For example, by considering the SGX implementation, it is easy to spot where the **createEnvironment** and **runAt** primitives are implemented, while **mapIn**, **MapOut**, **loadState** and **storeState** are implemented inside the SMM.

**A note on customizable data structures and algorithms.** In Algorithm 24 (line 16), we abused notation for brevity by inserting a call to the SMM to resolve a map miss. As the SMM appears to be inside the primitive, this may raise questions about how the user can customize data structures and algorithms in LaSt$^{GT}$.

We provide a straightforward alternative to customize them. The alternative is to extend the parameters of the execute primitive with a callback function for the SMM. If a callback is provided, the execute primitive uses it (at line 16) to call the user function. Otherwise the primitive can use a default implementation.

**Small modifications to the** verify **primitive.** Algorithm 25 and Algorithm 26 (dark background,

```
Input: description of code's text, data, stack sections, and input data (including nonce)
Output: output data
trigger registration hyper-call to isolate code and I/O memory pages
encode I/O parameters for I/O marshaling
call isolated code
decode output data
trigger unregistration hyper-call to return isolated memory pages
return output data
```

```
trigger registration hyper-call to isolate code and memory pages for input and output
trigger hyper-call to register input data root maps and SMM entry point
call isolated code
output data ← output data map
trigger unregistration hyper-call to return isolated memory pages
return output data
```

**Algorithm 23:** execute primitive on XMHF-TrustVisor. Above, the original implementation from Algorithm 1. Below, dark areas highlight the differences of the implementation that support LaSt$^{GT}$.

```
Input: description of code's text, data, stack sections, and input data (including nonce)
Output: output data
init SGX Enclave Control Structure (SECS)
initialize TCS(s)
trigger system call to ECREATE enclave
for each code page to be isolated do
    trigger system call to EADD enclave page
    trigger system call to EEXTEND the page content to the enclave
end for
trigger system call to EENIT the enclave
run EENTER on the enclave's TCS
forward local attestation to Quoting Enclave for remote attestation
trigger system call to EREMOVE the enclave
return output data
```

```
init SGX Enclave Control Structure (SECS)
initialize TCS(s)
trigger system call to ECREATE enclave
for each code page to be isolated do
    trigger system call to EADD enclave page
    trigger system call to EEXTEND the page content to the enclave
end for
trigger system call to EENIT the enclave
run EENTER on enclave's state handler's TCS to register input data root maps
run EENTER on the enclave's TCS
forward local attestation to Quoting Enclave for remote attestation
trigger system call to EREMOVE the enclave
return output data

On AEX:
    if map miss then
        call SMM
    end if
    do memory management (reclaim/provide pages)
    if map hit then
        run EENTER on state handler's TCS (and wait for termination)
        run ERESUME on enclave's TCS
    else
        segmentation fault ⇒ shutdown
    end if
```

**Algorithm 24:** execute primitive on Intel SGX. Above, the original implementation from Algorithm 2. Below, dark areas highlight the differences of the implementation that support LaSt$^{GT}$.

```
Input: nonce, output hash, input hash, code
    identity, hypervisor identity, hypervisor
    public attestation key, TPM attestation,
    certified TPM public attestation key,
    hypervisor attestation
Output: true or false
if TPM attestation can be validated using certified TPM public attestation key $\wedge$ attested hypervisor identity is expected $\wedge$ attested hypervisor public attestation key is expected $\wedge$ hypervisor attestation can be validated using hypervisor public attestation key $\wedge$ nonce, code identity, hash(input hash||output hash) are expected then
    return true
else
    return false
end if
```

```
Input: nonce, output hash,
    request hash and state (root) hash , code
    identity, hypervisor identity, hypervisor
    public attestation key, TPM attestation,
    certified TPM public attestation key,
    hypervisor attestation
Output: true or false
if TPM attestation can be validated using certified TPM public attestation key $\wedge$ attested hypervisor identity is expected $\wedge$ attested hypervisor public attestation key is expected $\wedge$ hypervisor attestation can be validated using hypervisor public attestation key $\wedge$ nonce, code identity, hash(request hash||state hash||output hash) are expected then
    return true
else
    return false
end if
```

**Algorithm 25:** verify primitive for XMHF-TrustVisor. Above, the original implementation from Algorithm 5. Below, dark areas highlight the differences of the implementation that support LaSt$^{GT}$.

```
Input: nonce, output hash, input hash, code
    identity, remote attestation (from Quoting
    Enclave), public report key
Output: true or false
if code identity (i.e., MRENCLAVE register value inside the remote attestation) not expected $\vee$ hash(nonce||input hash||output hash) (i.e., REPORTDATA structure inside the remote attestation) not expected then
    return false
end if
contact IAS [220] through APIs [221]
validate attestation report from IAS with public report key
if IAS returns attestation not valid then
    return false
end if
return true
```

```
Input: nonce, output hash,
    request hash and state (root) hash , code
    identity, remote attestation (from Quoting
    Enclave), public report key
Output: true or false
if code identity (i.e., MRENCLAVE register value inside the remote attestation) not expected $\vee$ hash(nonce||request hash||state hash|| output hash) (i.e., REPORTDATA structure inside the remote attestation) not expected then
    return false
end if
contact IAS [220] through APIs [221]
validate attestation report from IAS with public report key
if IAS returns attestation not valid then
    return false
end if
return true
```

**Algorithm 26:** verify primitive for Intel SGX. Above, the original implementation from Algorithm 6. Below, dark areas highlight the differences of the implementation that support LaSt$^{GT}$.

Note. Inside an enclave the verification is slightly different as follows: the signed Attestation Verification Report (returned by the IAS) is forwarded to the enclave, who performs the same checks as above and additionally verifies the signature, so it requires one additional input i.e., the public Report Key. For additional details, see the implementation of the get_cert primitive on SGX (Section 2.3.2).

|  | VC3 | Haven | LaSt$^{\mathrm{GT}*}$ |  |  |
|---|---|---|---|---|---|
|  |  |  | hypervisor | library | SQLite |
| SLoC<br>$\times$<br>$10^3$ | 9.2<br>[39] | 23.1+<br>$\mathcal{O}(10^3)^{\#}$<br>[38] | 15.1 [94]<br>+1.9$^{\ddagger}$ | 7.7 | 92.6 |
|  |  |  | 24.7 |  |  |
|  |  |  |  | 100.3 |  |

** based on XMHF-TrustVisor* ‡ LaSt$^{\mathrm{GT}}$ *core code and headers*

$^{\#}$ *LibOS contains millions of lines of code*

Table 4.2: TCB size breakdown (in thousand source lines of code) and comparison. SQLite is included as an example real-world application ported to LaSt$^{\mathrm{GT}}$.

bottom-side) highlight how the verify primitive changes. As it can be noticed, the main difference is providing the client with the state root hash, that allows to verify that the remote service processed the intended large data set. Therefore, the complexity of the primitive—and so the verification effort for the client—does not change with respect to the previous version (top-side of the algorithms).

## 4.6 Evaluation

We implemented LaSt$^{\mathrm{GT}}$ using XMHF-TrustVisor [53, 94], and now we analyze our implementation by quantifying its TCB, comparing it with the original XMHF-TrustVisor and running both micro-benchmarks and real-world applications.

**Experimental setting.** We use a Dell PowerEdge R420 Server equipped with: a 2.2GHz Intel Xeon E5-2407 CPU; 16GB of DDR3 memory; a TPM v1.2; a primary 300GB, 15Krpm hard-disk; a secondary 2TB 7.2Krpm hard-disk. The server runs Ubuntu 12.04 32-bit with a Linux kernel 3.2.0-27. The resources are fully dedicated to our experiments. LaSt$^{\mathrm{GT}}$ uses the secondary disk to ensure uniform experimental conditions.

Data is organized in chunks of 128MB and blocks of 256KB. Our micro-benchmarks justify these values for sequential workloads (typical of data analytics) and we suggest optimizations for random workloads (Section 4.6.5). We assume the worst-case scenario that requires LaSt$^{\mathrm{GT}}$ to maintain a low memory footprint. Hence, we configure the SMM to reclaim old chunk maps when the service code tries to access a new one.

### 4.6.1 TCB Size

We quantify LaSt$^{GT}$'s TCB size using the lines of source code (SLoC) metric, as calculated by the SLOCCount tool [195], and we compare it with previous work (Table 4.2). At the hypervisor level, the TCB size increases by 12%. The LaSt$^{GT}$ library adds an additional 7.7 SLoC of user-level code, which is relatively small. For example, it increases the size of the SQLite service code by only 8.3%.

The table also compares the TCBs of LaSt$^{GT}$, VC3 and Haven. Haven's TCB is notably large due to the library OS. LaSt$^{GT}$'s TCB is larger than that of VC3. However, first, LaSt$^{GT}$ currently includes XMHF-TrustVisor's TCB, while VC3 is based on SGX and so does not include any privileged code; second, LaSt$^{GT}$ is not application-specific and can run generic self-contained applications.

We expect LaSt$^{GT}$'s SGX implementation to have a smaller TCB than the current one. As SGX keeps privileged code out of the enclave, the hypervisor functionality—to manage the VMs, protect the isolated memory from the untrusted OS, and schedule the execution of trusted and untrusted applications—will be moved out of the TCB and implemented in untrusted code, while retaining similar security guarantees.

### 4.6.2 Comparing LaSt$^{GT}$ and XMHF-TrustVisor

As a baseline experiment, we compare LaSt$^{GT}$ with the original XMHF-TrustVisor implementation. LaSt$^{GT}$ can be much faster than XMHF-TrustVisor depending on the amount of data that the application processes, as displayed in Figure 4.10. The experiment uses a 512MB dataset that is read sequentially and fits in memory; XMHF-TrustVisor cannot process data whose size is larger than memory. The figure allows us to compare the time (y-axis) each tool takes to read some amount of data (x-axis). XMHF-TrustVisor always exhibits a large startup time (dependent on the data size) as it reads everything upfront into memory. In contrast, LaSt$^{GT}$ exhibits a performance that is related to the parts of memory that are actually read/written thanks to its ability to do incremental data loading and validation. For example, in an execution that only touches half of the dataset, TrustVisor would roughly end up taking twice as much time as LaSt$^{GT}$.

### 4.6.3 Microbenchmarks

LaSt$^{GT}$ incurs overhead when it needs access to additional data through page faults. The primary sources of overhead include switching control between software components, (un)loading maps in isolated memory, and disk accesses by the SMM. Since the last one is the same for trusted

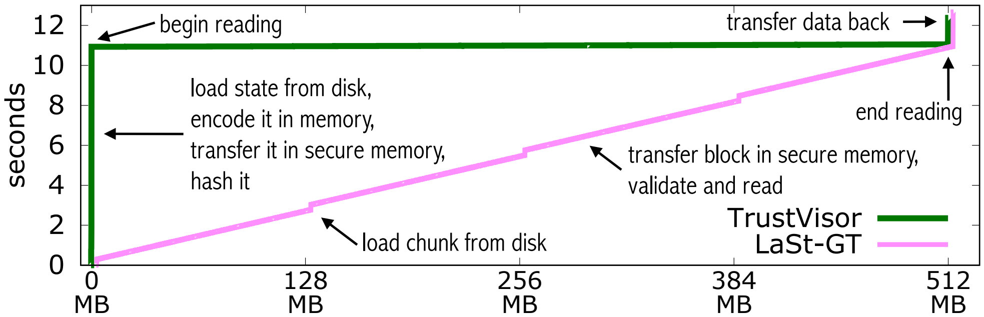


Figure 4.10: Comparison between LaSt$^{\text{GT}}$ and XMHF-TrustVisor.

and untrusted executions, we just focus on quantifying the overhead of the first two. We also quantify the overhead of preparing the state hierarchy by the content source. We present results that are the average of $1000$ experiments with a $95\%$ confidence interval.

**Context-Switching.** We measure the overhead to switch between the Supervisor, the state handler and the SMM (Section 4.4.3.2).

The Supervisor invokes the SMM when data is needed from disk. This involves switching from the trusted to the untrusted environment, and back (see table below). This time is mostly used to transfer the memory map list between the untrusted and the trusted execution environments. This requires inspecting the nested page tables of the virtual machine to check permissions for the data transfer, and then modifying both the nested page tables with the new permissions and the sensitive environment page tables to add (or remove) the pages from the isolated virtual address space (Section 4.5.3).

| ❶ trusted-untrusted env. switching | ❷ state handler resumption | ❸ SMM resumption |
|---|---|---|
| $191.68\mu s \pm 0.12$ | $36.11\mu s \pm 0.08$ | $39.93\mu s \pm 0.5$ |

Table 4.3: Context switch and application resumption overhead.

A second source of overhead is that associated with resuming the state handler, after some data has been brought into the secure environment, and the SMM for disk access. These resumption times (Table 4.3) include the overhead of the XMHF-TrustVisor software stack[1], of virtualization to resume the trusted VM or the untrusted VM, and of the VM interruption to return back into the hypervisor. The slightly higher and more variable overhead for the SMM can be attributed to scheduling delays (time slicing, preemption) caused by the OS. This does not occur in the isolated (and dedicated) execution environment where the state handler runs.

This means that the overheads for invoking the SMM and the state handler from the Super-

[1] TrustVisor is an application running within XMHF [94].

visor after a page fault are ❶+❸ and ❶+❷ respectively, in addition to the processing cost (e.g., accessing disk or validating data).

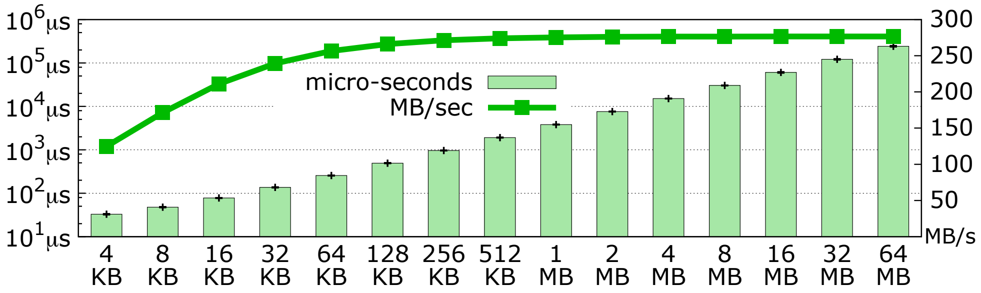


Figure 4.11: Log-scale average time (bars, left y-axis) and speed (line, right y-axis) for mapping data (x-axis) inside/outside the isolated trusted environment. The right y-axis shows the attained speed.

**I/O data mapping overhead.** The overhead for transferring memory maps between the two execution environments (Section 4.5.3) is shown in Figure 4.11. Larger maps can be transferred at higher speed. This suggests that if the application has to process all data in a map, it is advantageous to transfer more data per fault to reach high-speed (e.g., using a 256KB block size as we did).

**From user data to LaSt$^{GT}$-compatible state.** Figure 4.12 shows the cost of building the state hierarchy (Section 4.4.2) for user data sizes from 1MB to 2GB. This is sufficient for our evaluation since disk bottlenecks (for reading data and writing back metadata) show up already at 64MB. For larger states, the throughput stabilizes at ≈ 60MB/s.

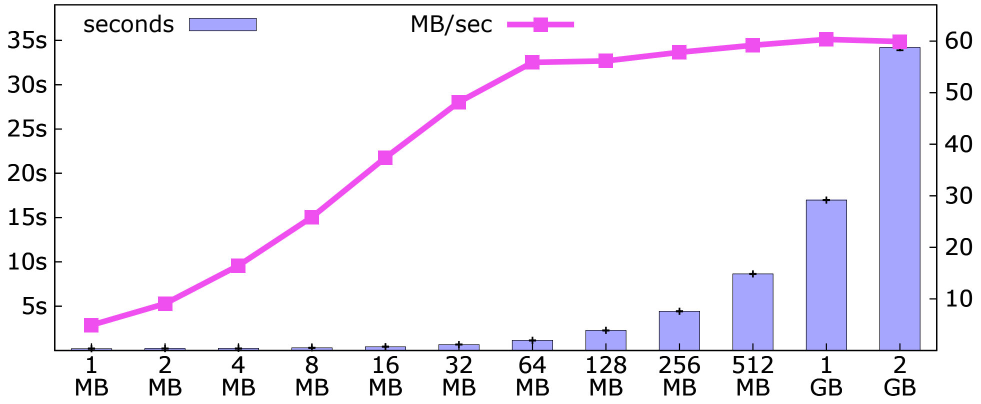


Figure 4.12: Time (bars, left y-axis) and speed (line, right y-axis) to build the LaSt$^{GT}$-compatible state for different state sizes.

Building the hash tree also incurs a high cryptographic cost. It needs to hash $2^9 \times 256$KB-sized blocks and $2^{10} - 2$ tree nodes (i.e., all nodes except the root). The procedure is optimized to take linear time in the size of the hash tree. We chose `SHA-256` as the hash function and carefully optimized it.

Different applications can leverage the incremental construction and parallelize the operations. We recall and stress in fact that chunks can be built separately.

### 4.6.4 End-to-End Application Performance

We run experiments with three applications with different data access patterns. They include: a simple application that sequentially walks 1TB of data, checking the first page of each data chunk; a nucleobase search application [126, 3.2] that accesses data sequentially, requiring all blocks of all chunks; and SQLite [188], which accesses random blocks of chunks.

**Tera-scale data processing.** We use a synthetic state of 1TB. The LaSt$^{GT}$-compatible state contains (in addition to the state root and one directory) one master chunk of 2.5MB that carries a list of 8192 chunks. This master chunk size is much larger than the 256KB hash list contained in the master chunk due to additional metadata (e.g., size and name) relative to the chunks that we maintain. Each chunk has 33KB of metadata (32KB due to the static hash tree, so 97%) and 128MB of data.

The execution environment is initially composed by a heap map of 262K memory pages (1GB) loaded lazily—though just a few are used in this application—and a state root map that fits in 1 page. Directory and IMELs are loaded as they are accessed, and they also fit in 1 page each. The master chunk instead fits in 641 pages. As chunks are accessed, two additional maps are included in the environment: the chunk metadata that fits in 9 pages and is loaded; and the chunk data that fits in 32768 pages and is lazily loaded. Only about 15 maps are present at a time due to the state hierarchy and the assumed environment constraints.

Figure 4.13 provides the progress of the application. It takes roughly 13 hours to process the terabyte of data, rather steadily at 23MB/s—the zoomed in segment shows a slight variability.

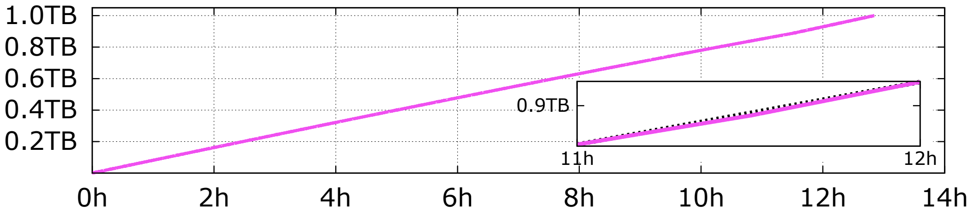


Figure 4.13: Progress in hours to process 1TB of data. In the zoom-in, the black straight dashed line highlights a negligible variability in processing speed.

We emphasize that the experiment uses a dataset between $1$ and $2$ orders of magnitude larger than previous work on secure data analytics [39, 125]. Also, it is unprecedented on the XMHF-TrustVisor software stack [53, 62, 94]—designed for small applications—and on a bare-metal hypervisor.

**Nucleobase search.** This application [126, 3.2] searches for a nucleobase sequence among the billion-scale reads (i.e., fragments obtained from DNA sequencing machines [207]) present in the FASTQ format[2] of the human genome in [208]. The application is relevant to investigate protein-coding mRNA sequence or to assemble sequences to reconstruct a contiguous interval of the genome [127].

Figure 4.14 shows the outcome of our experiment on a human genome of roughly $0.3$TB. The experiment produces about $1.15$ million page faults that are handled directly by the hypervisor, and about $2.24$ thousand are forwarded to the SMM for grabbing data from disk. The Nucleobase search is slower than the first application because it uses all the data and also involves more processing.

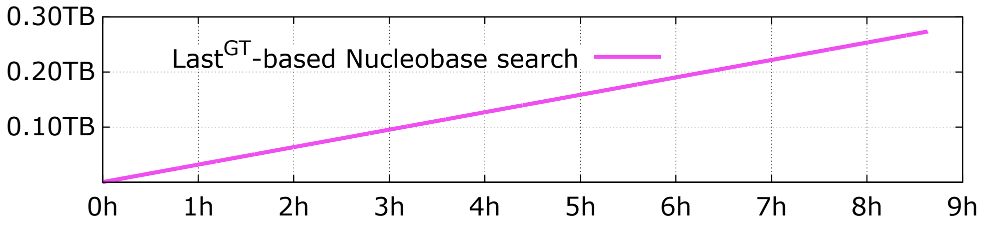


Figure 4.14: Progress in hours to process a $0.3$TB large human genome.

**Database engine.** The final application is the full SQLite (v.3.8.7.2) [188]—a real-world database engine with a non-trivial code base of $92.6$K lines of code—that we execute on LaSt$^{\text{GT}}$ without modifying its source code. We compile it with a virtual file system module that uses the LaSt$^{\text{GT}}$'s library to access the state, and with an abstraction layer that uses LaSt$^{\text{GT}}$'s functionality for memory management and I/O. The benchmark measures the time to query key-value stores of different sizes to get the value associated to a specific key.

The results are shown in Figure 4.15. The query time grows slowly until $x = 128$MB due to the larger data that is loaded in untrusted memory and to the larger hash tree that has to be validated in the trusted execution environment. At $x = 128$MB, $256$MB, $512$MB the query time stabilizes. This is due to the data access pattern of SQLite which only involves the first chunk. At $x = 1$GB, $2$GB however, SQLite also accesses the 7th chunk before going back to the first one.

[2] Common text format used for storing sequences and quality score.

This forces LaSt$^{\mathrm{GT}}$ to load and validate the metadata of several chunks and to maintain their data in untrusted memory in the case it is accessed. This happens similarly with larger databases. For instance, at $x = 0.25$TB, SQLite requires access to one more chunk (the 420th) in addition to those listed before. Hence, the overhead scales linearly with the number of accessed chunks.

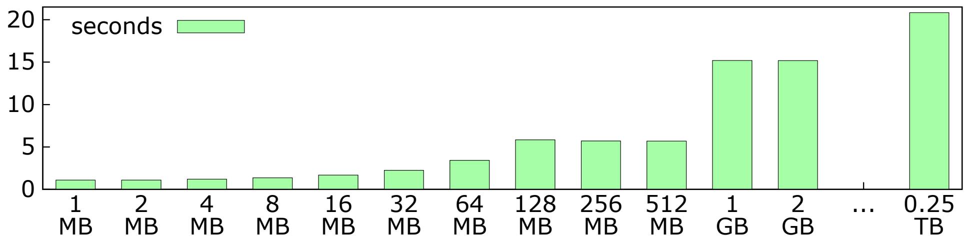


Figure 4.15: Time (y) to query a SQLite-based key-value store of different sizes (x). The state is built using 128MB chunk and 256KB block sizes.

**A glimpse of chunk & block size optimization.** In order to show the benefits of optimizing LaSt$^{\mathrm{GT}}$ for specific application requirements, we repeated the previous SQLite-based experiments by first protecting the database using a smaller chunk size (1MB) and block size (4KB). The results in Figure 4.16 show about an order of magnitude better performance. This is chiefly due to the smaller chunks read from disk and smaller data blocks transferred in secure memory whenever the engine performs a random memory access. The performance with a terabyte-scale state is in the order of a few seconds. This is due to a large master chunk that contains metadata for many (about 280K) small chunks, in contrast with the few (about 2K) large chunks in the previous experiment.

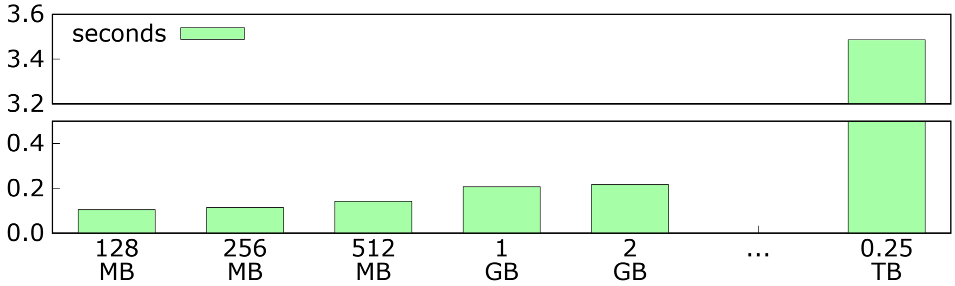


Figure 4.16: Time (y) to query a SQLite-based key-value store of different sizes (x). The state is built using 1MB chunk and 4KB block sizes.

### 4.6.5 Discussion

The performance of LaSt$^{\mathrm{GT}}$ can be optimized by tuning the parameters of state hierarchy and by leveraging better performing trusted components. First, the chunk and block sizes that we fixed for all experiments can instead be tuned for specific applications. This optimization problem is left for future work. However, intuitively (as we also evidenced in Section 4.6.4), smaller chunk and block sizes reduce unnecessary data validation and data loading, while they slightly increase the authentication metadata in the hierarchy. Second, when a memory map is created, LaSt$^{GT}$ optimizes for lazy loading in isolated memory, but XMHF-TrustVisor still requires the data to be present in untrusted main memory. This means writing data in memory bypassing critical optimizations such as lazy loading from disk. In this case, we believe that the SGX implementation can be beneficial to take advantage of the highly optimized kernel software stack, in addition to avoiding expensive virtualization operations such as VMEXITs and maintaining nested page tables.

## 4.7 Summary

This chapter shows that current solutions that provide hardware-based integrity guarantees for large-scale data applications make a trade-off between security and functionality, namely: they support unmodified generic applications but they have a large TCB; or they have a small TCB but they are application-specific (e.g., for MapReduce). In order to mitigate such trade-off, this chapter presents the design, implementation and evaluation of LaSt$^{GT}$, a secure system built on a generic trusted component for self-contained x86 code that processes large-scale data.

LaSt$^{GT}$ has a small TCB and enables end-users to verify the validity of the results received from the securely isolated application. Specifically, in line with the objective of this thesis, LaSt$^{GT}$ provides the following integrity guarantee to a client:

> *"if the client can verify the results attested by the trusted component on the service provider platform, then the client request was processed by the intended code on the intended (large) input state, so the received response can be trusted. "*

Also, LaSt$^{GT}$ clearly separates duties between software components: the service code performs the intended computation; a library (linked with the service code) provides access to large-scale data; a separate handler code guarantees the integrity of the data, and it is fully independent from the service code; finally, the privileged code running below supplies memory resources as required. This makes easy to build or port applications on LaSt$^{GT}$, and also to customize data structures and algorithms of our default implementation.

Finally, our experiments with applications such as databases and genome analytics prove that general large-scale applications can run on systems with a small TCB. We also provide evidence that the overhead in XMHF-TrustVisor is mostly due to expensive data I/O and context switch. We expect that such overhead can be heavily reduced by using Intel SGX and by optimizing LaSt$^{\text{GT}}$'s parameters according to the requirements of a specific application.

## Chapter 5

# Availability in Trusted Executions

Now we focus on how to make trusted executions available. Availability is about guaranteeing that a computing service can be accessed and used in spite of failures on the cloud provider's platforms, and it is one of the fundamental computer security objectives [128]. This is particularly relevant in the cloud computing model (Section 1.3.1) where security threats (Section 1.1) harm both the client, through service and data disruption, and the service provider by harming its reputation and profit.

The usual approach for providing availability is to use Replication [129]. In particular, making component replicas work together so that, when some components malfunction, others can keep the system operating correctly.

The key challenge in replication is how to coordinate these replicas. Our security model (introduced in Section 1.3.1 and extended later in Section 5.2.1) makes this challenge more difficult to deal with because it assumes that these replicas can behave arbitrarily. Such behavior is commonly known as Byzantine [130, 131]—due to service hijacking and vulnerabilities for instance. Hence, a replica can fail by crashing or by sending out forged messages with the intent of disrupting the functionality and data throughout the distributed system.

There are two main paradigms for designing a replicated system, namely *active replication* (AR) [132, 133] or *passive replication* (PR) [134, 135, 136, 137]. At a high level, AR makes multiple service computations, while PR replicates the state of a service. We provide a brief overview of these techniques.

In AR (Figure 5.1 left-side), all replicas run the service operations. PBFT [138] is a well-known example. In order for these replicas to return the same result following an execution, they are required to be deterministic (state machines). For this reason, AR is also commonly known as State Machine Replication (SMR). Also, in order for the replicas to provably reach and agree on

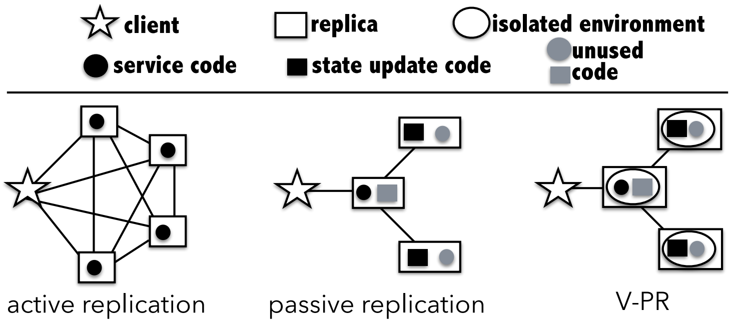


Figure 5.1: Comparison among replicated systems based on active replication (left), passive replication (middle), verified passive replication (right).

a common output despite arbitrary failures, it is assumed that at most a third of them can be faulty. Byzantine Fault Tolerant (BFT) protocols are known as BFT-SMR. Besides the overhead of executing redundant operations, these protocols require to run non-trivial coordination protocols (e.g., consensus or atomic broadcast) to maintain consistency in the distributed system. Although significant effort has been spent to make BFT-SMR practical [138, 139, 140, 141, 141, 142, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155], the cost continues to be high. Furthermore, BFT-SMR non-deterministic executions may cause the state on the replicas to diverge [154, 156, 157, 158]. The solution is often to assume that the service is deterministic, thereby restricting its applications.

In PR (Figure 5.1 middle), only the primary replica runs the service, while the backup replicas receive and apply state updates. PR delivers several benefits but has a significant security drawback. First, it saves CPU resources for computationally intensive services since only one replica executes the service code. Second, it allows replicating non-deterministic services since backup replicas do not have to replicate the same execution; rather, they catch up with the primary replica using the state updates. Third, the centralized control offered at the primary is resource-efficient and enables easy and cheap coordination of the other replicas without requiring consensus or atomic broadcast protocols. These reasons justify the attractiveness of the approach, which has been adopted by many popular systems, like GFS [159], Boxwood [160], Niobe [161], Remus [162], Tardigrade [163] and Zookeeper [164]. The security drawback is that it does not tolerate arbitrary failures since the computation is not replicated. In fact, a malicious primary could compromise the system state undetectably by hijacking the service execution flow and/or broadcasting forged updates to the backup replicas.

In this chapter we improve PR to bridge the security gap with AR. In particular, we introduce the concept of *Verified Passive Replication* (V-PR, Figure 5.1 right-side). V-PR is a fully passive replicated system that inherits the efficiency benefits of PR and uses our generic Trusted Computing component (TCC, Section 2.3) to provide security guarantees in a hybrid failure model. The model excludes physical or side-channel attacks on the TCC and is similar to that introduced in [139, 165, 166], since the trusted component can only fail by crashing, while the rest of the system can experience arbitrary failures; we elaborate more on the model in Section 5.2.1. By using the TCC to secure the service, V-PR ensures that the execution at the primary replica is verifiable through the identity attestation of the executed code. The backup replicas can thus establish trust in the state updates coming from the primary once they can verify that the intended code produced them. The primary replica can similarly establish trust in a backup replica, thereby ensuring that the state has been replicated correctly.

As a result, V-PR is both secure and efficient. Malicious software is unable to tamper with the operations performed at a replica, and in particular with the messages it sends out, which are authenticated in the trusted execution environment. The overall system is resource efficient for three reasons. First, it reduces the demand of computational resources, which is typical of PR systems. Second, the coordination protocol is cheaper (in terms of network messages) than typical AR protocols. Third, the TCC allows a decrease on the number of replicas with respect to AR schemes that do not leverage a trusted component. In addition, V-PR natively supports non-deterministic executions. In our experimental evaluation, we show the benefits of V-PR by comparing it with state-of-the-art replication protocols and by implementing a real-world database engine on top of it.

### Contributions

- We present the design of the V-PR system backed by a trusted component that always guarantees safety in a hybrid failure model, and also guarantees liveness during periods of partial synchrony.
- We define secure protocols for system initialization, fault-tolerant execution and primary change.
- We implement V-PR on XMHF-TrustVisor [53, 94]. We show that the performance of V-PR is close to that of state-of-the-art protocols in BFT-SMR such as BFT-SMaRt [167] and Prime [168]. We port the SQLite [188] database engine on V-PR. Finally, we provide evidence that V-PR's main source of overhead derives from the underlying trusted component, which will become more efficient as Trusted Computing technology continues to improve.

## 5.1 Overview of Verifiable Passive Replication

### 5.1.1 Rationale Behind Execution Verification in Replication

A service $\mathcal{S}$ may experience several types of failures that can undermine its correctness. Among these, Byzantine failures [131] represent the most general class. This means making no assumptions about the correctness of $\mathcal{S}$'s execution on a platform. As a result, the observable actions of $\mathcal{S}$ (i.e., the messages it sends out) are allowed to be arbitrary, so they cannot be trusted. For example, these messages can be: none, as if the platform crashed; corrupted, due to a bit that was flipped; or malicious, as if they were forged due to a software attack.

Byzantine failures represent an important issue in PR. Clients and replicas have no means for checking the execution of the service, and the computation performed by the primary is not replicated, thereby lacking redundancy. In this aspect, the problem turns out to be similar to that of checking the execution of services outsourced to untrusted clouds that we introduced in Chapter 1 and that is the subject of thesis.

### 5.1.2 Solution and Challenges

As we did for outsourced services, using our abstraction (Chapter 2.3) to devise solutions for efficient identification of large code (Chapter 3) and for large-scale data processing (Chapter 4), we solve this problem by leveraging our TCC for execution isolation, code attestation, secure storage and (additionally) trusted counters. This allows us to: reduce the attack surface, for instance by excluding the OS from the TCB; protect and identify the code executed at the primary; program the backups and clients to verify the execution of the primary replica; prevent rollback of the state of the replicas. Hence, we can avoid the AR approach, inherit the benefits of PR (particularly efficiency and non-deterministic executions) and provide clients with integrity and availability guarantees of the passively-replicated service execution.

There are however several challenges to be overcome.

- First, although backups can verify the primary's execution, the primary itself must also be able to ensure that its state (i.e., the data) is propagated to the backups as intended. If the data has not been replicated, the primary's crash can result into data loss. This means that our solution requires mutual verification between primary and backup replicas.
- Second, how can we efficiently initialize the distributed system? We need a way to bring the replicas into an operative trusted state in which they can communicate with each other efficiently. In Trusted Computing parlance, verification is about attestation, but attestations

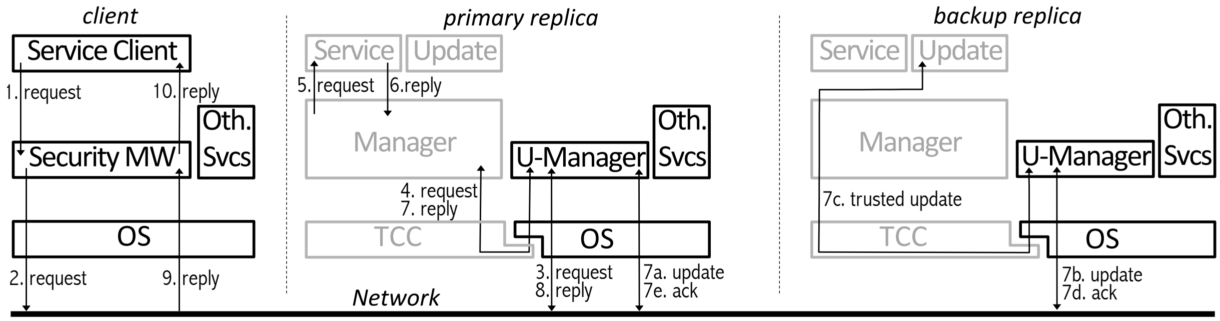


Figure 5.2: Architecture of V-PR. Light shaded parts correspond to the trusted system components.

are expensive. Therefore, we want to minimize their use.

- Third, how can we ensure that replicas cannot be rolled back, particularly the primary? Recall that we are assuming a hybrid failure model where the TCC (so the trusted execution environment) behaves correctly but the rest of the system can experience arbitrary failures, while previous PR models assume that the primary is trusted. So in our case, the state of the trusted service code could be rolled back to a previous version while it is stored in the untrusted environment.
- Fourth, how to prevent replies from being forwarded to clients before the state updates (if any) are propagated. Again, in PR this is not an issue because the primary is trusted and waits for acknowledgement from the backup replicas before replying. In our case however, once the untrusted part of the primary receives the replies from the trusted service code (i.e., when the execute primitive returns), it could forward these back to the clients immediately. Unfortunately, if the primary fails, this results into data loss because backup replicas are not up to date.

### 5.1.3 Architecture of V-PR

We now present V-PR, a fully-passive replicated system that allows the cloud provider to provide availability guarantees to clients. More precisely, V-PR is always safe in the hybrid failure model, while liveness is guaranteed during periods of partial synchrony (see Section 5.2.1). Figure 5.2 displays the architecture of V-PR. In each replica, we distinguish between the trusted execution environment (light shaded boxes on the left-side of each replica) and the untrusted execution environment (right-side of each replica). The client is fully trusted and can reach the primary replica through the network.

We distinguish between three layers in the distributed system. At the bottom, we have the TCC (Section 2.3.1) and the OS that provide support for executing service applications and for data I/O. In the middle, we have middleware components: the security middleware, or SMW, at the client; the Manager and U-Manager at the replicas. These components implement the core of V-PR. In addition, there may be other service applications, independent from V-PR, running at the client and at the replicas. At the top, we have the service client (at the client) and the service and update state application (at the replicas). The service application refers to application-level code for which clients require integrity and availability guarantees. The update application applies state updates received by the primary.

The core components of V-PR take care of security and availability. In particular, the SMW and the primary Manager establish a secure channel between the client and the primary (and so the replicated system). The channel is used to secure messages between the service client and service application. The Managers implement key management, authentication and the replication logic inside the trusted environment. The U-Managers instead are simple applications implemented on top of the untrusted OS to perform data I/O with the respective Managers, network and storage operations.

### 5.1.4 V-PR's Operations

As we mentioned, V-PR is a fully-passive replicated system and works as follows. A client sends a request to the primary. The primary executes the request (e.g., a query from the client over a database). If there are state updates, the primary broadcasts them to the backup replicas. When the primary receives enough acknowledgments, the reply is sent back to the client.

More into details, during the initialization of the system, the client's SMW securely exchanges a secret key with the primary's Manager. In addition, all the replicas' Managers share a secret key. These keys allow the establishment of a secure channel between the client and the primary and another secure channel among the replicas.

V-PR's operations can thus be described as follows (see Figure 5.2). The service client sends a request (1) to the replicated system using the SMW to authenticate it (2). The request is delivered by the OS to the primary U-Manager (3), and then forwarded to the primary Manager (4). The primary Manager validates the request and passes it (5) to the service application for the execution. The primary Manager retrieves the reply (6) and gathers the state updates, if any. Both the reply and the updates are protected and transferred to the primary U-Manager (7) in the untrusted environment. The primary U-Manager broadcasts the updates to the backup replicas

| Features | Replication Protocols | | |
|---|---|---|---|
| | AR | PR | V-PR |
| Byzantine Resistant | **yes** | no | **yes**<br>except physical failures |
| Replicas | **2f+1**<br>with trust assumptions | **2f+1** [169]<br>f+1 [134] | **2f+1**<br>with trust assumptions |
| Asynchronous<br>for safety, partially synchronous for liveness | **yes** | **yes** [169]<br>no [134] | **yes** |
| (Re-)Computations | $\mathcal{O}(n)$ | $\mathcal{O}(\mathbf{1})$ | $\mathcal{O}(\mathbf{1})$ |
| Message Size | $\mathcal{O}(\mathtt{d}_{\mathtt{in}})$ | $\mathcal{O}(\mathtt{u})$ | $\mathcal{O}(\mathtt{u} + \mathtt{d}_{\mathtt{out}})$ |
| Non-determinism | no | **yes** | **yes** |

Table 5.1: Comparison between Active Replication (AR), Passive Replication (PR) and our new proposal Verified Passive Replication (V-PR).

(7a), where these are delivered to the backup U-Manager (7b). The updates are then forwarded to the backup Manager who validates and passes them to the update application that applies them deterministically, and finally returns an acknowledgement to the backup U-Manager (7c). The backup U-Manager forwards the acknowledgement back to the primary (7d), where it is delivered to the primary U-Manager (7e). After enough acknowledgements are received and processed by the primary, the authenticated reply is sent back to the client (8). At the client, the reply is delivered to the SMW (9), validated and then passed to the service client application (10).

### 5.1.5 Benefits and Drawbacks of V-PR

We discuss how V-PR combines benefits and drawbacks of AR and PR (Table 5.1).

- V-PR is safe in the hybrid failure model. In particular, it is robust against arbitrary failures that do not concern the TCC, for example: software attacks, compromised OS, message corruptions, physical attacks that do not affect the trusted component (e.g., disk data snooping and corruption). Also, it can tolerate crash failures that concern the TCC and its execution environment, for instance: a crash of the service code application, a hardware failure that makes the TCC halt.

  However, it does not protect against programming flaws in the service. This holds even in SMR systems unless the replicas employ diverse service software [170, 171]. Arguably, given the minimality and constraints of our trusted execution environment, such latent vulnerabilities could be more difficult to exploit in V-PR.

  V-PR also does not protect against persistent or transient hardware failures in the trusted component that disrupt the service (unless it crashes). Persistent failures may also affect SMR systems that do not use diverse hardware. Instead, transient failures could be addressed through functional hardware redundancy. For example, the Recovery Unit in an IBM z10 maintains the whole processor state in a buffer to retry the work on error; also, instruction-processing damage checks are performed [172]. So this is an orthogonal issue.

- V-PR is resilient-optimal. It only requires the correctness of a majority of replicas, which is a lower bound for crash-tolerant asynchronous distributed systems [173]. Similarly, this also holds for AR systems that leverage a trusted component, thereby making trust assumptions. In the absence of such a trusted component, the optimal resiliency requires two thirds of the replicas to be correct [131]. The PR system in [134] uses fewer replicas but assumes the synchronous model.
- V-PR is safe in the asynchronous model, though it requires partial synchrony for liveness [173, 174]. This ensures that the system keeps working despite the failure of the primary replica. The requirement also holds for several SMR systems [151, 167, 175, 176], since assuming partial synchrony allows circumventing the impossibility of coordinating replicas in a completely asynchronous system [177].
- Computational efficiency and simplicity are achieved by relying on a single service execution. This saves overall a linear factor in processor cycles and request ordering effort with respect to AR. In fact, previous SMR systems [140, 149, 178] require several executing replicas that are linear in the number of faults they tolerate. The saving also holds for the number of exchanged messages because Byzantine agreement requires a number of messages that is quadratic in the number of replicas [179] (assuming optimal-resiliency).
- The size of the messages that the replicas exchange is application-dependent. In AR, this depends on the request size (i.e., input data $\mathtt{d}_{\mathtt{in}}$), while in PR it depends on the size of the state updates $u$. V-PR additionally includes the client reply $\mathtt{d}_{\mathtt{out}}$ alongside the state updates to ensure that it is delivered even upon the failure of the primary.

  Another difference is related to the number of messages exchanged with the client. In AR, such number is usually a (linear) function of the number of replicas in the system, since the client has to make sure that enough replicas performed the work correctly. In PR, instead, only the primary exchanges messages with the client. In V-PR specifically, this is safe since the client can verify that he exchanges messages with the service code that runs in the trusted execution environment of the primary replica.

- V-PR can run a non-deterministic service, similar to PR. This is an important difference with respect to AR, whose theory is based on "deterministic" SMR. Some complex techniques are known to relax such deterministic behavior assumption. One is to guarantee that replicas reach the same state and output after non-deterministic operations using the execute-verify-recover paradigm [154], while another technique is to hide the non-determinism using abstraction [180].

## 5.2 V-PR: Verified Passive Replication

This section presents the replication model to clarify the assumption we make for V-PR (Section 5.2.1). Also, it introduces the V-PR context structure for storing security-sensitive local information and explains how V-PR interacts with the TCC to protect it (Section 5.2.2). Then it describes how the V-PR system is initialized (Section 5.2.3), how it works in a normal execution (Section 5.2.4) and how it deals with the failure of the primary (Section 5.2.5).

### 5.2.1 Replication Model and Hybrid Failure Model

We extend the model introduced in Section 1.3.1 with some additional assumptions that are specific to V-PR.

The system consists of a group of $n$ replicas, each one equipped with a TCC. The V-PR Manager and the replicated service run in the protected environment provided by the TCC. The rest, including the V-PR U-Manager, the OS and other services, execute in the untrusted part of the replica. For a background about the TCC and its primitives we refer to Section 2.3.1. An arbitrary number of clients can access the system, but they need to be enrolled beforehand to obtain the necessary authentication credentials (e.g., by contacting an identity management service).

At most a minority $f = \lfloor \frac{n-1}{2} \rfloor$ of the replicas can fail. At each replica, the TCC can only suffer crashes but the rest of the system and network might experience arbitrary (or Byzantine) failures. This means that code running in the trusted execution environment either produces correct results or no values. Untrusted components (such as the U-Managers) may corrupt data, delay the execution or do any other attempt to maliciously break the protocol. With regard to messages, they might be modified, removed or delayed during transmission.

The presented model is similar to the hybrid failure model introduced in [139, 165, 166] and to those used in previous work based on trusted components [139, 140, 141, 151, 152, 181, 182]. The similarity lies in the trusted hardware component that is assumed to behave correctly. The main difference lies is the power of such a component, which performs computation by means of the execute primitives, while in previous work it is mainly used to assign trusted counter values to messages and to sign them in order to prevent equivocation (due to malicious processes) throughout the system.

V-PR ensures safety in the hybrid failure model, and asynchrony can only prevent the replicas from making progress. Liveness is guaranteed in periods of partial synchrony, when messages are delivered and processed within a fixed but possibly unknown time bound. This can be achieved through retransmissions and acknowledgments.

### 5.2.2 Securing V-PR's Context using the TCC

As the execute primitive runs code in the trusted environment to process a request and terminates when a reply is available, a secure mechanism is necessary to store local V-PR information (i.e., the current state, whether the replica is the primary or a backup, any secret key already exchanged with the other replicas). The execute primitive in fact has not been designed to run indefinitely, waiting for more requests to arrive. Also, no direct access to persistent storage is available from the trusted execution environment in order to minimize the trusted computing base. Hence, the local V-PR information must be provided as input from the untrusted environment, validated and protected by the Manager and returned as output to the U-Manager for long-term storage, waiting for the next execution.

In V-PR, the local information of a replica is stored in the context data structure ($ctx$). The TCC secure storage primitives allow protecting and validating $ctx$ as it transits between the trusted and the untrusted execution environments, as part of the input and output of the execute primitive. Also, the TCC trusted counters prevent such data structure to be rolled back by code in the untrusted environment while trusted code is not running.

**The Context Data Structure** is constituted by the following most relevant fields:

- $id$: the identifier of the replica
- $nreplicas$: the number of replicas
- $clientCred$: for client authentication
- $state$: a description of the current service state, i.e., $\mathsf{h}(state^j)$. It identifies the state unambiguously when associated with a trusted state counter, i.e., the pair $(state\ counter, state)$ is unique
- $K$: a system-wide shared key, created by the primary during the initialization and forwarded to the backups

- $auth$: authenticator to protect the $ctx$ structure while it is stored by the U-Manager (e.g., a MAC)

**How V-PR leverages TCC secure storage.** The integrity of $ctx$ and the confidentiality of $K$, which is shared only among Managers in the trusted environment, are critical for the system's operations. They are protected by each Manager before returning to the U-Manager: (i) an authenticator (e.g., a MAC) is computed with $K$; (ii) $K$ is encrypted using the auth_put TCC primitive, specifying the identity of the running trusted application code (which includes the code of the Manager, the service and the update applications) as the intended receiver; consequently, $K$ cannot be accessed from the untrusted environment by the U-Managers, nor by any application executing on the TCC with a different identity. Later on, when the Manager is re-executed, it calls the auth_get primitive to decrypt $K$, and then verifies the integrity of the received context. We will omit these steps while presenting the protocols.

**How V-PR leverages trusted counters.** The TCC maintains a *trusted state counter* and a *trusted view counter* between executions of the Manager. The Manager handles them using the create_cnt, get_cnt and incr_cnt TCC primitives (Section 2.3.1). The former counter is used to assign unique and ordered values to state updates (and consequently also to state versions) and it is incremented whenever client requests produce a modification of the service state. The latter counter is used to determine the role of a replica at a given time, by counting the number of primary changes: a replica is the primary if $viewcounter \bmod nreplicas = id$, otherwise it is a backup. These trusted counters are important in the case of malicious replicas to prevent state rollback attacks and to prevent the existence of multiple primaries that can make progress, because they could produce divergent states.

### 5.2.3 System Initialization

The system initialization must ensure three key objectives: (i) every correct replica of the system is able to join the group, while malicious replicas are excluded; (ii) a system-wide secret key $K$ is securely shared among the replicas, so that messages can be exchanged securely and efficiently among the trusted execution environments of the replicas; (iii) all replicas finish the initialization with the same starting state ($state^1$). The main challenge in solving this problem is that the storage primitives only work with the local hardware component (Section 2.3.1), so the Managers do not have an immediately available secure channel, and this has to be built using the insecure U-Managers. PR does not need them because replicas are assumed to be trustworthy, so a more lightweight mechanism is sufficient.

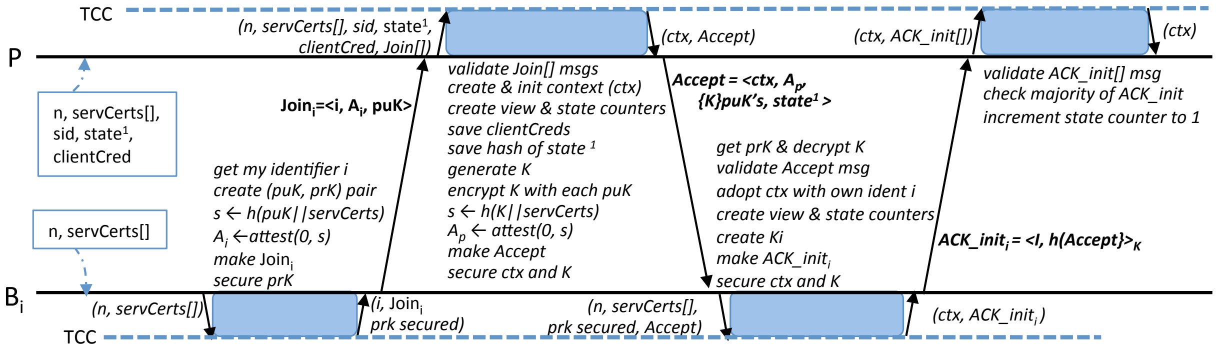


Figure 5.3: V-PR system initialization. The first two messages (i.e., the first $B_i \to P$ and $P \to B_i$) are protected through attestation while the other messages (i.e., the second $B_i \to P$) use a MAC authenticator based on $K$.

The initialization protocol is displayed in Figure 5.3. When primary and backup Managers start, their identities are computed. Then they run the initialization protocol inside the trusted execution environment, leveraging the TCC primitives for secure storage and trusted counters. The untrusted U-Managers simply take care of message passing among replicas and with their associated Managers.

The first objective is achieved in two steps. In the first step, the system administrator supplies the replicas with the size of the group and a vector of TCC certificates. Each certificate belongs to one specific replica's TCC. In addition, the primary replica is also given a service identifier, the initial service state ($state^1$) and a set of credentials ($clientCred$) to authenticate clients. In the second step, the Managers at the backup replicas begin to run a protocol for mutual attestation with the primary Manager. Each backup Manager generates a fresh public/private key pair and performs an attestation of the public key—while the private key is locally saved in secure storage—and the certificate vector. The attestation allows the primary Manager to verify the identity of the trusted application running at the backup replica, to establish trust in the public key generated by the backup Manager and to ensure that the replicas are setting up a secure group. A $Join$ message is finally returned and sent to the primary U-Manager with the backup replica identifier, the attestation and the public key.

Notice that the identity of the code running in the trusted environment can be easily verified because all replicas are implemented with the same trusted code. So they have the same code identity when they are loaded and executed by the TCCs. They differentiate among themselves through the index in the certificate vector that points to the public key of the local TCC.

The second objective is attained by means of a secure symmetric key exchange between the primary and the backup Managers. The primary Manager validates the $Join$ messages, i.e., the attestation of each backup replica, its public key and the certificate vector. Also, it creates and initializes a new context $ctx$. In particular, it saves the (hash of the) initial service state, a newly generated random secret key $K$, and it creates the trusted view and state counters with the create_cnt primitive (Section 2.3.1) and identifiers $sid||1$ and $sid||2$ respectively. Then, the primary Manager encrypts $K$ using the public key of each replica, and finally attests $K$ and the certificate vector. This later allows the backup replicas to verify the identity of the primary, to establish trust in $K$ and to ensure they are setting up a secure group. An *Accept* message is then returned and sent to the backup U-Managers with the new context, the attestation, the encrypted $K$ and the initial service state. Finally, each backup Manager validates the *Accept* message, including the primary's identity, decrypts $K$ and initializes its local context and counters.

The third objective is achieved by making each of the backup Managers send an acknowledgment ($ACK\_init_i$) to the primary, authenticated with $K$. The primary Manager waits for the arrival of enough ACKs in order to install the initial service state ($state^1$) and advance the state counter. This state is then propagated to the backup Managers through the initial state updates, as described in the next section.

### 5.2.4 Normal execution

V-PR's processing provides the following guarantees in the hybrid model (Section 5.2.1): (i) client requests are served as intended; (ii) state updates are installed; (iii) recoverability and consistency are ensured in the case of failures. Read requests are only handled by the primary since they do not update the state. For write requests, the procedure requires a majority of replicas to be available. Also, on the critical path, it uses three sequential trusted executions: two at the primary Manager for processing a request and the respective acknowledgments, and one at the backup Manager to process the state updates.

The description below assumes that client and the primary Manager share a key $Kcl$. Depending on the client authentication solution, $Kcl$ may be derived on-the-fly by combining the identifier ($clientID$) and the credentials ($clientCred$) supplied by the system administrator, or through a key distribution protocol. For example, $clientCred$ could correspond to a master key $K\_master\_cl$, whose confidentiality is protected with the TCC secure storage; $K\_master\_cl$ could be used to derive client specific keys $Kcl = KDF_{K\_master\_cl}(clientID)$, where $KDF$ is a secure key derivation function; during the enrollment process in the system, the client could be given an identifier $clientID$ and $Kcl$. It should be noted that $clientCred$ is included in the context ($ctx$) structure; as $ctx$ is used by all replicas, any replica can recover $Kcl$ when it becomes the

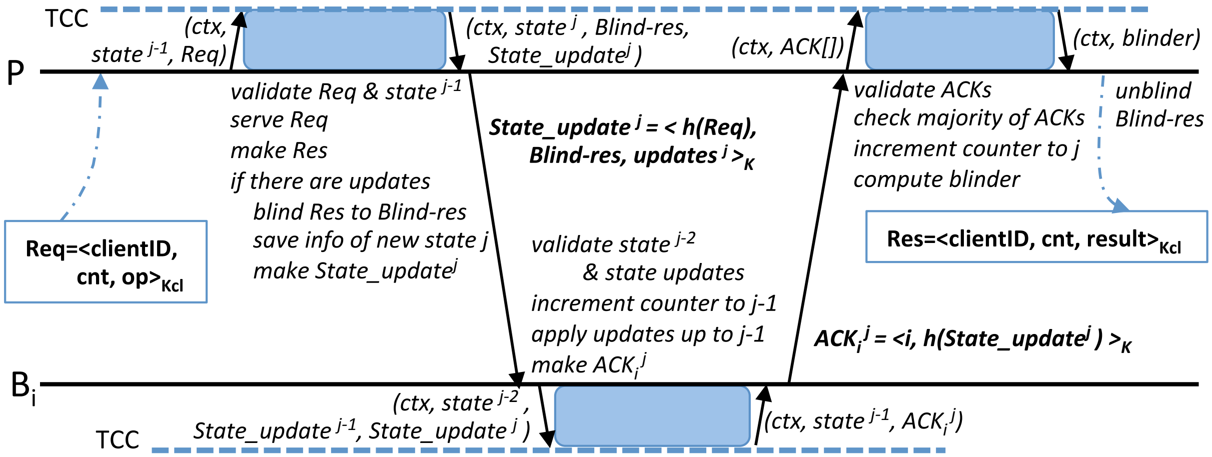


Figure 5.4: V-PR normal case operations to serve a client request that produces state updates.

primary. As another example, the technique in Section 3.3.5.1 could be used to let the client and the primary exchange a symmetric secret key, by leveraging the client's fresh public key and the primary's attestation to verify the primary's code and establish trust in the exchanged key. The interactions between the client and the primary Manager can then be protected with this shared key using standard solutions.

Figure 5.4 shows how V-PR processes requests. Whenever the client calls a service operation, it creates a request message ($Req$) with the $clientID$, a session counter ($cnt$) and an operation ($op$) including the associated parameters. The message is authenticated and integrity protected with $Kcl$ (e.g., by adding a MAC), and transmitted to the primary.

When the message is delivered, the primary U-Manager executes the Manager with the last context ($ctx$), the current state version ($state^{j-1}$) and the request. The Manager validates the context, the request and the state, and runs the operation. Next, it produces a response message ($Res$) for the client using the output returned by the service. The response to a read request is simply authenticated and sent back to the client.

Instead, the response to a write request raises the challenge of propagating the updates to the backups before the client receives a response. This challenge is specific to V-PR because in hybrid failure model the U-Manager cannot be trusted to propagate the updates, to wait for the acknowledgements and then to send the reply to the client. Let us describe the issue using the following example scenario: a client requests to store some data $d$; the primary Manager performs the operation and returns an authenticated success response; the U-Manager does not broadcast the state updates, but rather forwards the response immediately to the client; now let us assume that the primary replica crashes and another one takes over; the client is unable to read $d$ back since the data is lost.

Our solution is to blind the response ($Blind\text{-}res$). With the blinding procedure the primary Manager ensures that the client cannot accept the response in the case the U-Manager forwards it too early by skipping the state propagation step. This is achieved by XORing a blinder value with the response message authenticator, thereby preventing the validation of the message. The blinder value is computed using a secure pseudo-random number generator seeded with the secret $K$, the state counter and the current state hash. This associates the blinder value to the state that has to be propagated and that was generated together with $Res$.

In addition to the blinded response, the primary Manager also releases an updated context and the state updates. In particular, in the updated context, the state field is the hash of the updated state (after the write request request has been performed). Also, a $State\_update^j$ message is created for the backups; this includes the client request, the blinded reply and the update information ($updates^j$) to bring the state from version $j-1$ to version $j$. The hash of the request and reply are useful in case of failure to match the retransmitted client request to its associated response, without re-doing the operation in $state^j$. If these are not available, the new primary proceeds from $state^{j-1}$.

Backup Managers do not explicitly send messages to each other to coordinate. We leverage the trustworthiness of the primary Manager to securely make this optimization. In particular, the arrival of $j$-th update at a backup implies that state $j-1$ was accepted by a majority of (primary and backup) replicas, and that the primary therefore decided to proceed with state $j$. The result is that backup replicas do not need to send messages to each other to ensure that the updates have been received and match. Rather, they can infer this fact when they receive the subsequent update from the primary. In turn, this means that they do not immediately apply an update.

When $State\_update^j$ is received, backup Managers must validate the $State\_update^{j-1}$ and $State\_update^j$ messages, in order to check for corruptions and ensure that they are applied in the correct order [1]. In particular, in order to apply the updates in $State\_update^{j-1}$, the trusted state counter must be equal to $j-2$, which indicates the last state that was applied, even by the other replicas. In this case, the $j-1$ update is applied and the trusted state counter is incremented to $j-1$. Also, each backup Manager $i$ creates an acknowledgement $ACK_i^j$ to inform the primary Manager that $State\_update^j$ is locally available and has been validated, though not yet applied.

The primary U-Manager waits for the arrival of a majority of these acknowledgements. Then, it executes the primary Manager, who checks for a majority of valid acknowledgements—this en-

[1] Notice that for the first state update $j$ is equal to 2; $State^0$ is the uninitialized service state, and $State\_update^1$ is actually represented by the initial state from the initialization procedure.

sures that at least one correct replica has received the latest state that survives in the case of multiple failures. The primary Manager then increments the state counter to $j$, thereby installing the last state, and outputs the blinder value. The primary U-Manager finally unblinds the blinded response message, by simply XORing the blinder value to the message authenticator, and forwards it to the client who can now validate it.

### 5.2.5 Fault Handling

V-PR handles failures of the primary by changing the primary though, due to the peculiar failure model (Section 5.2.1), using a different mechanism than PR's. In particular, the Managers do not execute continuously and have no direct access to network resources—they run on the TCC with the execute primitive (Section 2.3.1)—so they cannot perform system/network monitoring. In PR instead, there are no such constraints since the replicas are trustworthy.

In V-PR, the recovery procedure is timeout-driven and is triggered by the U-Managers. This allows the backup replicas to select a new primary in order to make progress. Although U-Managers are considered untrusted, recall that a majority of U-Managers are assumed not to mount a DoS attack, such as not executing the recovery protocol or executing it frequently. This is reasonable since it is in the interest of the cloud provider that the system keeps operating to benefit the clients. Therefore, each U-Manager is assumed to begin the recovery procedure when it stops receiving valid state updates for a period to time[2].

The selection of a new primary is guided by two principles: *Unique Majority* and *Progress Evidence*. First, a primary is effectively changed when a majority of replicas increment their trusted view counter to the same value. This guarantees the uniqueness of a primary that is able to make progress, since a majority of the backups recognize its authority to issue state updates. This also forces the other replicas to move to the newer view because client requests, or updates, cannot be successfully processed in the older view.

Second, a replica can safely apply all state updates up to state $j-1$ when state update $j$ is received, independently of the view number. In fact, for the primary Manager to issue the state update $j$, it must have received the acknowledgements for state $j-1$. Consequently, replicas may change view, but they can still make progress with any two consecutive updates. This is safe because, as we mentioned, the primary Manager has to receive acknowledgments from a

[2] The U-Managers, however, should adjust the timeouts to reflect the current situation where the primary is taking longer to transmit messages (e.g., due either to processing delays or network congestion). To prevent the case where a malicious primary U-Manager tries to delay the whole system, the timeouts are only increased up to a certain value defined by the system administrator.

majority of replicas for the $j-1$ update in order to later issue the $j$ update.

In more detail, the recovery protocol encompasses the following steps. All backup replicas start the timeout. When the timeout at a backup replica expires, the U-Manager executes the local Manager to obtain a $Probe\_change$ message. The message contains an authenticated description of the current configuration (including the state and view counter values) and it is broadcast to all replicas.

Replicas wait for a majority of $Probe\_change$ messages that match the same configuration of their local Manager before moving to the next phase. In the meanwhile, they continue to participate in the system as usual, in case more recent state updates are delivered.

The Manager is called again when enough $Probe\_change$ messages are delivered. It validates the messages and then returns a $Probe\_reply$ also containing the current configuration. A set with a majority of valid $Probe\_reply$ messages that match the same backup Manager's configuration triggers the view change. The Manager is called to increment the trusted view counter and a final $New\_primary$ message is produced and broadcast to inform about the new view.

**A corner case.** Let us assume that the old primary crashes while broadcasting the $j+1$-th state update ($u1$). If the new primary receives this message, then it re-broadcasts the same update to be processed by backup replicas. When enough acknowledgements arrive, the client reply—recall that this is included in the update message—is unblinded and forwarded to the client.

However, if the new primary does not receive the state update that was issued by the previous primary replica, then the client times out and re-issues the request. The new primary can therefore make progress by computing a new $j+1$-th state update (say $u2$ and possibly $u2 \neq u1$). Since the backup replicas have not yet increased their trusted state counters to $j+1$, they can favor $u2$ over $u1$ and drop $u1$. In fact, this update will only become definitive when the $j+2$-th update is broadcast. Then the computation proceeds as in the normal case (Section 5.2.4).

## 5.3 Experimental Evaluation

This section provides details on the implementation of V-PR and analyzes its performance in a cluster of servers. We evaluate V-PR and compare it with two publicly available open-source state-of-the-art libraries that implement BFT-SMR, namely BFT-SMaRt [167] and Prime [168]. Both libraries are being actively maintained[3] and have been used in several other studies, e.g., [155, 176, 183, 184, 185, 186].

[3] at the time of writing

### 5.3.1 Implementation

**Trusted execution environment.** We instantiated our TCC using XMHF-TrustVisor. The trusted hypervisor does not natively support the trusted counter primitives. We have implemented them following the same approach used for the existing sealed storage primitives. Trusted counters can be accessed only if the running code—whose identity is saved by the hypervisor—satisfies the access control policy (i.e., it is the same one that initially created the counters). We refer to Section 2.2.1 for some background about XMHF-TrustVisor and to Section 2.3.2 for additional details on the implementation of the primitives.

**Message passing.** V-PR (more precisely each U-Manager) uses a communication subsystem based on ZeroMQ [112], a high-performance messaging library that abstracts details of the underlying socket implementation.

**Service application.** The service we replicate is a full SQLite database engine [188]. Our SQLite version is self-contained (i.e., with statically linked libraries and no OS support) to run in the isolated environment provided by XMHF-TrustVisor.

We had to address the challenge of how to allow SQLite to access the data and store it durably. In the isolated environment, in fact, the application does not have access to disk. We note that the durability property on a single platform is not strictly necessary in our case, because the data is also replicated on the backups—many of which are assumed not to fail. So we addressed this challenge by implementing a module for fast in-memory database operations. This is a Virtual File System (VFS) module that is registered inside SQLite, and allows the Manager to intercept updates to the state whenever SQLite modifies the database. We stress that we did not modify the SQLite code as the database abstracts the underlying implementation of the file system through the VFS notion.

This implementation, and V-PR more in general, is orthogonal to LaSt$^{\text{GT}}$ (Chapter 4) and it is currently not based on it[4]. The intercept of the updates is performed inside the VFS module which lies between SQLite and the low-level primitives for file I/O that are implemented in the LaSt$^{\text{GT}}$ library. This makes V-PR compatible with LaSt$^{\text{GT}}$. An interesting alternative would be to extend LaSt$^{\text{GT}}$ with V-PR, by adding functionality for tracking and propagating page writes in virtual memory. This would make the replication fully application-agnostic and usable by other LaSt$^{\text{GT}}$'s applications. We reiterate however that neither the integration with, nor the extension of, LaSt$^{\text{GT}}$ have been further investigated.

**Experimental setup.** Our testbed is a set of Dell PowerEdge R420 rack servers, running Ubuntu

[4] Also, V-PR [37] is prior to LaSt$^{\text{GT}}$ [41].

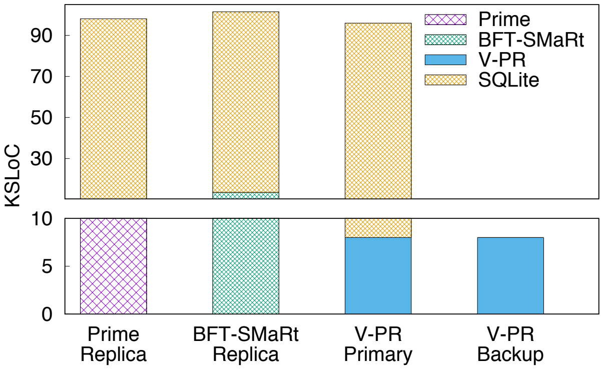


Figure 5.5: Actively-executed application-level code size of Prime, BFT-SMaRt and V-PR. The V-PR case is divided in two parts: the V-PR primary that actively executes the service code, and the V-PR backup that only execute the update service.

12.04 with a Linux kernel version 3.2.0-27. The servers are equipped with Intel Xeon E5-2407 CPUs, 3GB of memory and a TPM v.1.2. We use the minimal number of machines that are required to tolerate one fault (i.e., $f = 1$), namely: three machines ($2f + 1$) are used to test V-PR; instead, BFT-SMR libraries are tested on four replicas ($3f + 1$), as their models require. The machines are connected with a $1$ Gb/s network through a Dell PowerConnect 5448 switch.

### 5.3.2 Analysis

This section addresses the following points: (1.) it quantifies the code size of V-PR and compares it against the size of Prime and BFT-SMaRt; (2.) it compares the basic performance of V-PR with Prime and BFT-SMaRt; (3.) it analyzes V-PR's overhead by running a simple zero-overhead service; (4.) it analyzes non-deterministic executions in V-PR; (5.) it evaluates the application-level CPU savings; (6.) it studies V-PR's end-to-end latency in a realistic scenario; (7.) it presents V-PR's main sources of overhead.

**1. Code size.** In Figure 5.5 we present a breakdown of the code size (in thousand source lines of code, KSLoC) of V-PR, BFT-SMaRt and Prime calculated with the SLOCCount tool [195]. For V-PR, we show the size of the core logic executed at the replicas, including the Manager, the U-Manager and the communication support built with ZeroMQ. The reported lines of code of BFT-SMaRt and Prime correspond to the respective releases that we downloaded. Notice that we consider the overall code size, and not the trusted computing base (TCB), because BFT-SMaRt and Prime do not execute code in a trusted environment.

We make the following observations. First, it is possible to notice that the size of the three replication schemes—considering the V-PR Primary implementation—is comparable and this is

| | BFT-SMaRt | Prime | V-PR |
|---|---|---|---|
| Messages ( r ) | 1+1 | | 1+1 |
| (w) | 4+3+16+16+4 | 3+16+16+3+12+16 | 1+3+3+1 |
| Hops | (r) 2 (w) 5 | 6 | (r) 2 (w) 4 |
| Replicas ($f = 1$) | 4 | 4 | 3 |
| Executions | 4<br>(active) | 4<br>(active) | 1 (active)<br>2 (passive) |

Table 5.2: Normal request execution in BFT-SMaRt, Prime and V-PR.

expected. However, when considering the code actively used at a backup V-PR replica the difference is significant. We clarify that the full V-PR and SQLite must be available at all replicas but, in the case of backups, the replicas never run the SQLite code while they remain backup replicas. So about $90\%$ of the total code is unused. In fact, backup replicas only need the SQLite code to ensure that the identity of the code in the trusted environment matches the code identity at the other replicas. This level of asymmetry is not possible in BFT-SMR because the correctness of the result stems from the replicated execution of the service. Even in the fault-free scenario, SMR has to guarantee that at least a majority of replicas execute the service [140], so they must have the code. Also, such asymmetry is relevant, and in favor of V-PR, when considering alternative system designs based on AR and trusted components.

**2. Basic performance comparison.** We analyze inherent features of V-PR, BFT-SMaRt and Prime at run-time (Table 5.2). The exchanged messages measure the coordination overhead to maintain state consistency, as shown in [138]. Each message exchange phase counts as one hop. V-PR outperforms BFT-SMaRt and Prime because it does not implement expensive coordination protocols for request ordering, such as Consensus, by verifying the work performed by each replica.

In V-PR, the primary does not have any interaction with the backups for read requests—client request and reply (i.e., 2 hops) are the only messages. V-PR recognizes automatically *a-posteriori* which requests contain read or write operations by tracking the changes to the database using the VFS module. BFT-SMaRt has a similar optimized execution for read-only operations—using the $invokeUnordered$ primitive—which avoids the call to the atomic multicast protocol. The main difference with respect to V-PR is that such capability has to be explicitly programmed (*a-priori*) into the client application by calling a specific read-only operation, while in V-PR this is not required. For Prime, although client operations can be classified as read-only or read/write, no such optimization is mentioned in [168] and we noticed no significant difference with respect to write-only requests—later on the authors confirmed our analysis.

Write requests must be ordered in all systems. In V-PR this procedure is centralized at the primary: client requests are ordered through a session counter, while state updates are ordered through the state counter. As a consequence, in a write request, the update message of the primary and the acknowledgements of the backups are the only messages that are transmitted in the distributed system. Compared to read requests, this results into 2 additional hops. In BFT-SMaRt and Prime, instead, the ordering procedure is an expensive Byzantine fault-tolerant protocol that requires several phases and messages exchanges among all replicas.

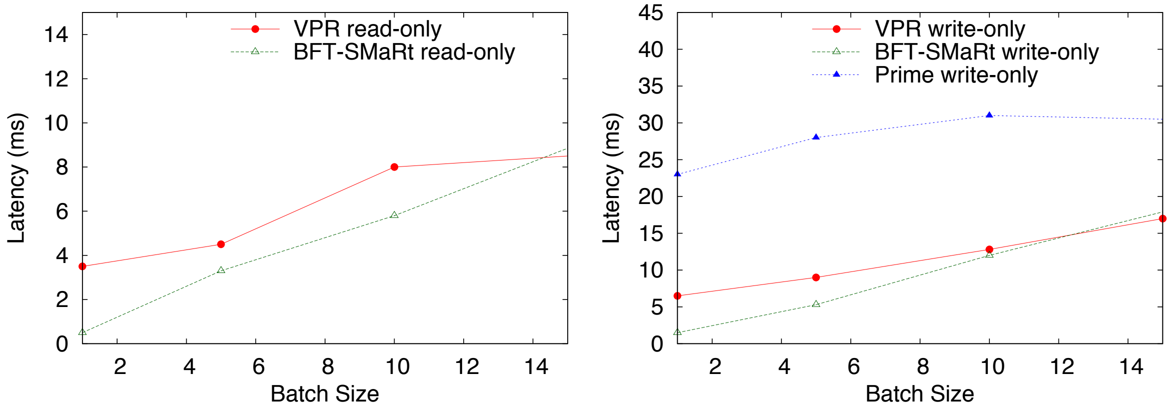


Figure 5.6: End-to-end latency, measured at the client, of a replicated zero-overhead service.

**3. V-PR's overhead with a zero-overhead service.** Figure 5.6 presents the results of an experiment based on a simple zero-overhead service in fault-free runs. This allows us to determine the replication overhead while excluding any overhead due to the replicated application. The results show that V-PR is slower than BFT-SMaRt for single requests, which is primarily due to the TCC. Also, Prime is mainly delayed by the heavy use of signatures. However, all systems take advantage of request batching[5] to improve their efficiency. Noticeably, V-PR matches BFT-SMaRt latency when the batch size reaches around 12 requests.

**4. Non-deterministic executions.** V-PR does not need to track non-deterministic actions during the execution of SQLite. As V-PR is based on passive replication, its target is the replication of the state that is the result of the execution after SQLite terminates; it does not target the actions that led to a specific state. Consequently, the primary does not need to track calls such as `sqlite3_randomness` as in Eve [154], or to implement upcalls as in BASE [180].

**5. CPU savings.** Figure 5.8 compares the application-level CPU consumption (in cycles) between a passive replication and an active replication of SQLite. Backup replicas do not execute SQLite. SQLite write operations are intercepted before they actually modify the database, so to form the state updates to be forwarded to the backup replicas. The backup replicas merely apply the state

[5] V-PR uses batching to send more data in a single call to the trusted Manager executing on the TCC.

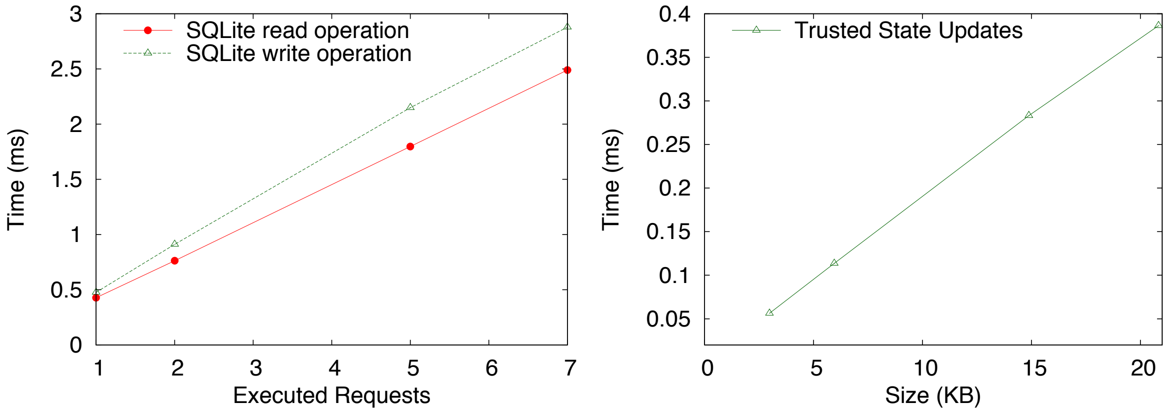


Figure 5.7: Application-level execution time of: read/write requests at the primary (left-side); state updates per-kilobyte at the backups (right-side). The time interval between execution start and termination is taken from the hypervisor.

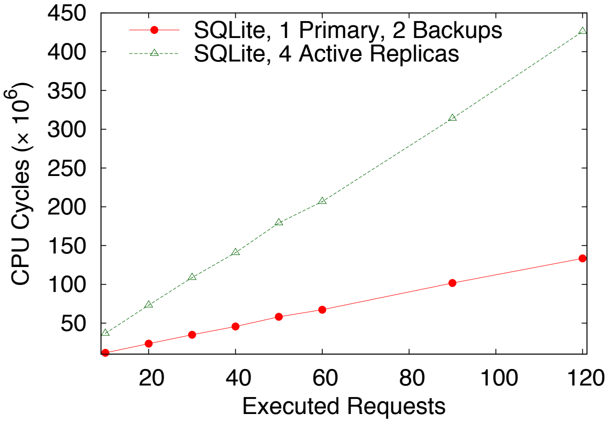


Figure 5.8: Average system-wide application-level CPU cycle consumption for passively and actively replicated SQLite deployments.

updates received by the primary. This is a simple deterministic procedure, whose complexity is linear in the size of the updates.

The amount of savings is proportional to the processing effort performed by the replicated service. In the experiments, the write workload is created with simple delete queries, that produce state updates and thus force V-PR to use the three replicas—recall that read queries are only executed at the primary. As we grow the number of delete requests, it is possible to observe an increasing gap between the two curves. More expensive queries, such as grouping and sorting operations, would make the gap wider, thus being favorable to V-PR.

**6. End-to-end measurements.** Figure 5.9 presents the performance of our V-PR-ed SQLite. We executed read-only requests (such as `select` operations) and write-only requests (such as `delete`

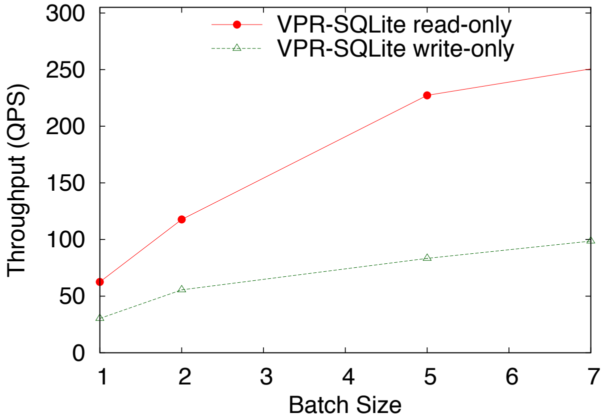


Figure 5.9: End-to-end performance of a V-PR-ed SQLite implementation.

operations) over a table of $200$ items. Writes are slower than reads due to the additional required TCC calls. Request batching allows to increase the throughput by more than $3\times$ for both types of operations, since it amortizes the TCC overhead (e.g., for loading and identifying SQLite code and input data) over multiple requests.

**7. Main sources of overhead.** The V-PR-ed SQLite overall incurs three costs, namely: (1) the SQLite and the update service execution costs, which have been shown in Figure 5.7; (2) the cost of the V-PR middleware (i.e., Managers and U-Managers), which has been shown in Figure 5.6; (3) the cost due to the underlying TCC. Since Figure 5.9 combines these costs together, we can argue that (3) is the most significant, and we also show how it can improve with newer technology.

As an example, let us consider the execution time of a single read request. SQLite takes about $0.5ms$ to complete, while V-PR's latency adds around $3.5ms$. Assuming that the TCC has zero cost, it follows that the expected throughput should be around $250qps$. Yet, the observed throughput is about $60qps$, i.e., $> 4\times$ lower than expected. A similar analysis holds for write requests and large batch sizes. Consequently, such high overhead is chiefly due the TCC that is based on XMHF-TrustVisor.

This is not surprising since the hypervisor performs several operations with non-negligible cost. In particular, it has to load and identify the application's code (including V-PR and SQLite); it has to load the input data and to provide memory for the output state updates. Additional costs include invoking the trusted counter and secure storage operations—we do not include the one-time attestation cost that is paid at setup time. These costs involve virtualization operations such as context switches and/or updates to the extended page tables for data I/O.

There are at least two ways to lower these costs. The first one is to batch the requests, the updates and the replies, as we have shown previously. This allows us to amortize context switches, code identification and input state loading over multiple executions. In fact, although batched data requires additional memory pages, the overhead for handling them is much smaller than that of serving a request in a separate trusted execution.

The second one is to leverage better performing Trusted Computing technology, which can be easily integrated below V-PR. In fact, V-PR is not bound to XMHF-TrustVisor but rather to the TCC abstraction (Section 2.3.1) that we implemented using the trusted hypervisor, and for which we also described an implementation based on Intel SGX [189]. SGX is expected to significantly improve the performance of V-PR with and without batching, because these optimizations are implemented above our TCC interface which is agnostic to the used trusted hardware component. In particular, V-PR would not incur any virtualization overhead and it would benefit from dedicated instructions on the fast CPU for trusted executions.

## 5.4 Summary

V-PR is the first fully-passive replication protocol that is safe and live in the hybrid failure model in the presence of partial synchrony. V-PR leverages trusted hardware to deliver security guarantees and to allow replicas to verify each other's execution. As a result, only the primary replica runs a (possibly non-deterministic) secured service, thereby saving computational resources at the backup replicas. Our experimental evaluation shows that: V-PR has comparable performance with state-of-the-art BFT-SMR protocols; it is cheaper than BFT-SMR protocols in terms of messages and number of replicas that are necessary to tolerate the same number of failures; its main source of overhead is due to our TCC implementation based on XMHF-TrustVisor. Our expectation is that V-PR can improve using a better performing TCC, for instance based on Intel SGX.

We remark that V-PR is orthogonal to, and can greatly benefit from: 1) our multi-identity approach (Chapter 3) to reduce and identify more frequently the code to be loaded inside the trusted execution environment; 2) our approach to handle a large state (Chapter 4) as it provides with an efficient and scalable data I/O mechanism. In addition, V-PR enables cloud providers to deliver highly available trusted services to clients, who can also efficiently verify that service results were produced and replicated as intended.

# Chapter 6

# Conclusions

Cloud computing—and more generally the use of remote resources—is becoming more and more attractive, but its adoption raises severe security issues. The problem stems from not owning and controlling the physical resources to run outsourced services. Clients and service providers are left with two alternatives: either to have faith in the cloud provider or to avoid using external resources.

In this thesis we argue that there is a third more satisfactory way out of this dilemma: to use trusted hardware to secure these outsourced services. Trusted hardware allows a cloud provider to execute an outsourced service while being unable to control its execution. In particular, the hardware provides with a root-of-trust that uses execution isolation mechanisms to prevent untrusted hardware and software to tamper with the code executing in the trusted environment.

Between such a simple concept and a concrete secure system for outsourced services there exist many challenges that are raised by the interests of different parties: hardware manufacturers constantly improve the technology and push for its adoption; cloud providers want to attract customers by offering diverse high-performance computational resources where they can run services; service providers do not want to limit the functionality of their services or to re-engineer them; clients want to have simple means to verify remote executions and to make trust decisions.

These challenges can be tackled starting from a Trusted Computing abstraction. The abstraction relies on a hardware root-of-trust and to provide a secure foundation for executing services. We show that a handful of primitives can fulfill the needs of all parties: manufacturers can build hardware that allows the implementation of the abstraction; cloud providers can offer hardware resources and a system where the abstraction allows to run services securely; service providers can use the hardware-agnostic abstraction to develop their services; clients can establish trust in the results they receive in one call. The contributions of this thesis bolster these claims and show

how to provide security and efficiency by working around these primitives.

We show how to deliver higher integrity guarantees and performance for large outsourced services by reducing the active TCB. The technique allows to load less code (w.r.t. a large service code base) in the execution environment and so to identify it more frequently. Since code identification is related to execution integrity, multiple identifications for shorter executions reduce the window of vulnerability with respect to a single identification for a longer execution. In addition, if a small amount of code is necessary for an execution, this can also result into better performance.

We show how to extend functionality to large-scale data processing using our abstraction and no calls for data I/O. The key for this goal is a clever use and management of virtual memory, which is commonly available in today's platforms. In particular, being "virtual", it does not require to expose any mechanism to the application via the abstraction interface.

Finally, we show that even implementing and running a highly available outsourced service is feasible and convenient. A distributed protocol allows to set up a distributed system by initializing service replicas. Then, by leveraging the trustworthiness of our hardware-based secure foundations, we can implement a passively replicated service. This benefits the cloud provider by saving computational resources, and also the service provider by allowing the implementation of non-deterministic services. Interestingly, even in this case, clients can maintain the one-call-to-verify feature.

## 6.1 Future Work

We outline interesting research topics that represent a natural continuation of this thesis work.

### 6.1.1 Additional implementations

First of all, it would be interesting to implement our findings on SGX. Although simple implementations could be attempted on current CPUs, we mention that LaSt$^{GT}$ is not yet implementable. The advanced virtual memory management features used by LaSt$^{GT}$ are in fact part of SGXv2 which, to the best of our knowledge, is not yet available on current hardware—only SGXv1 is available.

An additional implementation could be based on AMD SEV. As SEV targets virtual machines, the implementation would likely follow an approach very similar to XMHF-TrustVisor to reduce the TCB and enable trusted executions. Differently from XMHF-TrustVisor however, the implementation would not require a TPM, or to trust additional hardware modules, since the secure execution would be confined on the main CPU. In addition, the implementation would allow us to understand whether our TCC abstraction holds up or requires adjustments.

### 6.1.2 Combining our techniques together

Although we have shown that the presented techniques are orthogonal, they have not been integrated into one system. A future implementation however would likely borrow from our multi-PAL approach and from V-PR to extend LaSt$^{\text{GT}}$ further, rather than implementing them as they have been described. The reason is that the multi-PAL approach requires making suitable code modules, while V-PR requires additional code to intercept state updates performed by the applications. These requirements could be similarly handled by LaSt$^{\text{GT}}$ by tracking accesses to code and state pages in memory. This would further enhance the generality of our approaches.

### 6.1.3 Dynamically linked libraries

Making applications to be self-contained, as we assume in this thesis, requires putting together libraries from different developers. While this is feasible, it has some drawbacks. For instance, the libraries must be available at build time. Also, they are difficult to be updated for two reasons: first, they are statically linked in the executable; second, the update of a single bit changes (with high probability) the identity of the self-contained application, which cannot be verified by a client—unless the client is informed of the new identity.

We believe that software components from different developers should be identified and verified separately. Ideally, developers would implement their services/libraries and publish them together with a signature over their code identities for authentication and accountability.

### 6.1.4 Architecture-agnostic code identification

Although this thesis makes service development architecture-agnostic (through our abstraction in Chapter 2), different implementations of the abstraction may compute different identities for the executed code. For example, let us consider our self-contained SQLite database engine's code. When this service is executed and attested (using execute and attest, Chapter 2), the attested code identity is $id'$ in XMHF-TrustVisor, $id''$ on Intel SGX and (with high probability) $id' \neq id''$. So, at verification time, a client has to verify an identity that depends on the used architecture, despite the same piece of code is executed. Hence, service developers have to provide these trusted identities for different architectures, because code identity is not architecture-agnostic. Promising approaches to solve this problem are: a level of indirection through the creation of some loader code that performs a standardized identification of the executed code; or an attestation mechanism that conveys properties rather than the identity of a specific binary code [187].

### 6.1.5 Multicore trusted executions

Exploiting multicore architectures is highly desirable for performance, but it raises challenges, namely: how to schedule multiple threads securely while maintaining a small TCB; how to support thread creation, execution and coordination with other threads while avoiding to enlarge the interface.

# Bibliography

[1] Amazon, "Amazon EC2," *http://aws.amazon.com/ec2/*.
(cited on page 1.)

[2] Rackspace, "Cloud Servers," *http://www.rackspace.com/cloud/servers/*.
(cited on page 1.)

[3] Microsoft, "Microsoft Azure," *https://azure.microsoft.com*.
(cited on page 1.)

[4] IBM, "IBM Cloud," *http://www.ibm.com/cloud-computing/*.
(cited on page 1.)

[5] A. F. Simpao, L. M. Ahumada, J. A. Gálvez, and M. A. Rehman, "A Review of Analytics and Clinical Informatics in Health Care," *Journal of Medical Systems*, vol. 38, no. 4, p. 45, apr 2014.
(cited on pages 1 and 81.)

[6] K. Srinivas, B. Rani, and A. Govrdhan, "Applications of Data Mining Techniques in Healthcare and Prediction of Heart Attacks," *International Journal on Computer Science and Engineering (IJCSE)*, vol. 02, no. 2, pp. 250–255, 2010.
(cited on pages 1 and 81.)

[7] D. H. Chau, C. Nachenberg, J. Wilhelm, A. Wright, and C. Faloutsos, "Polonium: Tera-Scale Graph Mining and Inference for Malware Detection," in *Proceedings of the SIAM International Conference on Data Mining (SDM)*, apr 2011, pp. 131–142.
(cited on pages 1 and 81.)

[8] L. Akoglu, H. Tong, and D. Koutra, "Graph based anomaly detection and description: a survey," *Data Mining and Knowledge Discovery*, vol. 29, no. 3, pp. 626–688, may 2015.
(cited on pages 1 and 81.)

[9] K. Ren, C. Wang, and Q. Wang, "Security Challenges for the Public Cloud," *IEEE Internet*

*Computing*, vol. 16, no. 1, pp. 69–73, jan 2012.
(cited on pages 1 and 81.)

[10] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. Durbin, "The Sequence Alignment/Map format and SAMtools." *Bioinformatics (Oxford, England)*, vol. 25, no. 16, pp. 2078–9, aug 2009.
(cited on pages 1 and 81.)

[11] B. Langmead and S. Salzberg, "Fast gapped-read alignment with Bowtie 2," *Nature methods*, vol. 9, no. 4, pp. 357–359, 2012.
(cited on pages 1 and 81.)

[12] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina, "Controlling Data in the Cloud: Outsourcing Computation without Outsourcing Control," *Proceedings of the ACM workshop on Cloud computing security (CCSW)*, pp. 85–90, 2009.
(cited on page 1.)

[13] W. Jansen and T. Grance, "Guidelines on Security and Privacy in Public Cloud Computing," *NIST Special Publication*, pp. 800–144, 2011.
(cited on page 1.)

[14] J. Tang, Y. Cui, Q. Li, K. Ren, J. Liu, and R. Buyya, "Ensuring Security and Privacy Preservation for Cloud Data Services," *ACM Computing Surveys*, vol. 49, no. 1, pp. 1–39, jun 2016.
(cited on page 1.)

[15] W. R. Claycomb and A. Nicoll, "Insider Threats to Cloud Computing: Directions for New Research Challenges," in *Proceedings of the 36th Computer Software and Applications Conference (COMPSAC)*, jul 2012, pp. 387–394.
(cited on page 2.)

[16] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th Conference on Computer and Communications Security (CCS)*, 2009, p. 199.
(cited on page 2.)

[17] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *Proceedings of the Conf. on Computer and Communications Security (CCS)*, 2012, p. 305.
(cited on page 2.)

[18] M. S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cache Attacks Enable Bulk Key Recovery on the Cloud," in *Proceedings of the 18th Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2016, pp. 368–388.

(cited on page 2.)

[19] A. Haeberlen, P. Kouznetsov, and P. Druschel, "PeerReview: Practical accountability for distributed systems," in *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2007, pp. 175–188.

(cited on page 2.)

[20] A. Sherman, A. Stavrou, J. Nieh, A. D. Keromytis, and C. Stein, "Adding Trust to P2P Distribution of Paid Content," in *Proceedings of the International Conference on Information Security (ISC)*, vol. 5735, Berlin, Heidelberg, 2009, pp. 459–474.

(cited on page 2.)

[21] P. England and M. Peinado, "Authenticated Operation of Open Computing Devices," in *Proceedings of the 7th Australian Conference on Information Security and Privacy (ACISP)*, jul 2002, pp. 346–361.

(cited on pages 2 and 21.)

[22] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman, "A Trusted Open Platform," *Computer*, vol. 36, no. 7, pp. 55–62, 2003.

(cited on pages 2, 21, 49, and 51.)

[23] R. Rivest, "On databanks and privacy homomorphism," *Foundations of Secure Computation*, pp. 168–177, 1978.

(cited on pages 2 and 22.)

[24] M. Bellare, V. T. Hoang, and P. Rogaway, "Foundations of Garbled Circuits," in *Proceedings of the International Conference on Computer and Communications Security (CCS)*, oct 2012, pp. 784–796.

(cited on page 3.)

[25] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: outsourcing computation to untrusted workers," in *Proceedings of the 30th Annual Conference on Advances in Cryptology (CRYPTO)*, aug 2010, pp. 465–482.

(cited on page 3.)

[26] A. C. Yao, "Protocols for secure computations," in *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (SCFS)*, nov 1982, pp. 160–160–164–164.

(cited on page 4.)

[27] ——, "How to generate and exchange secrets," in *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. IEEE, oct 1986, pp. 162–167.
(cited on page 4.)

[28] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish, "Verifying computations with state," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 341–357.
(cited on page 4.)

[29] B. Parno and C. Gentry, "Pinocchio: Nearly practical verifiable computation," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2013, pp. 238–252.
(cited on page 4.)

[30] J. van den Hooff, M. F. Kaashoek, and N. Zeldovich, "VerSum," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, nov 2014, pp. 1304–1316.
(cited on page 4.)

[31] Y. Zhang, J. Katz, and C. Papamanthou, "IntegriDB," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, oct 2015, pp. 1480–1491.
(cited on pages 4 and 83.)

[32] D. Fiore, C. Fournet, E. Ghosh, M. Kohlweiss, O. Ohrimenko, and B. Parno, "Hash First, Argue Later," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016, pp. 1304–1316.
(cited on page 4.)

[33] D. Sgandurra and E. Lupu, "Evolution of Attacks, Threat Models, and Solutions for Virtualized Systems," *ACM Computing Surveys*, vol. 48, no. 3, pp. 1–38, feb 2016.
(cited on page 4.)

[34] Trusted Computing Group, "TPM Main Specs v1.2, Rev. 116," 2011.
(cited on pages 4 and 7.)

[35] IBM, "IBM PCI-e Cryptographic Coprocessor," *http://www-03.ibm.com/security/cryptocards/pciecc*.
(cited on pages 4 and 7.)

[36] S. Bajaj and R. Sion, "TrustedDB," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2011, pp. 205–216.

(cited on pages 4, 8, and 52.)

[37] B. Vavala, N. Neves, and P. Steenkiste, “Securing Passive Replication Through Verification,” in *Proceedings of the 34st IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2015, pp. 176–181.

(cited on pages 4, 18, 52, and 134.)

[38] A. Baumann, M. Peinado, and G. Hunt, “Shielding applications from an untrusted cloud with Haven,” in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, oct 2014, pp. 267–283.

(cited on pages 4, 8, 12, 24, 49, 52, 78, 81, 82, and 107.)

[39] F. Schuster and M. Costa, “VC3: Trustworthy data analytics in the cloud,” in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015, pp. 38–54.

(cited on pages 4, 8, 17, 24, 81, 83, 107, and 112.)

[40] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, M. L. Stillwell, D. Goltzsche, D. Eyers, P. Pietzuch, and C. Fetzer, “SCONE: Secure Linux Containers with Intel SGX,” in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 689–704.

(cited on pages 4, 8, 12, 24, and 83.)

[41] B. Vavala, N. Neves, and P. Steenkiste, “Secure Tera-scale Data Crunching with a Small TCB,” in *Proceedings of the 47th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, jun 2017.

(cited on pages 4, 17, 24, and 134.)

[42] B. Parno, J. M. McCune, and A. Perrig, “Bootstrapping Trust in Commodity Computers.” in *IEEE Symposium on Security and Privacy*, 2010, pp. 414–429.

(cited on page 4.)

[43] S. W. Smith and S. Weingart, “Building a high-performance, programmable secure coprocessor,” *Computer Networks*, vol. 31, no. 9, pp. 831–860, 1999.

(cited on page 6.)

[44] J. Winter and K. Dietrich, “A Hijacker’s Guide to the LPC Bus,” in *Proceedings of the 8th European conference on Public Key Infrastructures, Services, and Applications*, 2012, pp. 176–193.

(cited on page 7.)

[45] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza, “AsyncShock: Exploiting synchronisation bugs in intel SGX enclaves,” in *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, vol. 9878 LNCS, 2016, pp. 440–457.
(cited on page 8.)

[46] Y. Xu, W. Cui, and M. Peinado, “Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems,” in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, may 2015, pp. 640–656.
(cited on page 8.)

[47] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, “Preventing Page Faults from Telling Your Secrets,” in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIACCS)*, 2016, pp. 317–328.
(cited on page 8.)

[48] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, “Software Grand Exposure: SGX Cache Attacks Are Practical,” *arXiv:1702.07521*, feb 2017.
(cited on page 8.)

[49] J. Seo, B. Lee, S. Kim, M. Shih, I. Shin, D. Han, and T. Kim, “SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs,” in *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, 2017.
(cited on pages 8 and 24.)

[50] M.-w. Shih, S. Lee, K. Taesoo, and M. Peinado, “T-SGX : Eradicating Controlled-Channel Attacks Against Enclave Programs,” in *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, 2017.
(cited on pages 8 and 24.)

[51] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, “Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu,” in *Proceedings of the Asia Conference on Computer and Communications Security (ASIACCS)*, 2017, pp. 7–18.
(cited on page 8.)

[52] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, “Flicker: An Execution Infrastructure for TCB Minimization,” in *Proceedings of the European Conference in Computer Systems (EuroSys)*, 2008, pp. 315–328.
(cited on pages 8, 30, 52, 53, 56, 77, 78, 81, and 82.)

[53] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, “TrustVisor: Efficient

TCB Reduction and Attestation." in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2010, pp. 143–158.

(cited on pages 8, 23, 25, 27, 30, 37, 38, 51, 52, 53, 56, 70, 78, 81, 82, 107, 112, and 119.)

[54] A. M. Azab, P. Ning, and X. Zhang, "SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms," in *Proceedings of the 18th Conference on Computer and Communications Security (CCS)*, 2011, pp. 375–388.

(cited on pages 8, 23, and 52.)

[55] R. Strackx and F. Piessens, "Fides," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, oct 2012, p. 2.

(cited on pages 8, 23, and 52.)

[56] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.

(cited on pages 8 and 83.)

[57] R. Floyd, "Assigning meanings to programs," *Mathematical aspects of computer science*, 1967.

(cited on page 9.)

[58] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM (CACM)*, vol. 12, no. 10, pp. 576–580, oct 1969.

(cited on page 9.)

[59] E. W. Dijkstra, "Structured programming," in *Chapter I: Notes on Structured Programming*, O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Eds. Academic Press Ltd., 1972, pp. 1–82.

(cited on page 11.)

[60] B. Liskov and S. Zilles, "Programming with abstract data types," *ACM SIGPLAN Notices*, vol. 9, no. 4, pp. 50–59, apr 1974.

(cited on page 11.)

[61] A. S. Tanenbaum, "Lessons learned from 30 years of MINIX," *Communications of the ACM (CACM)*, vol. 59, no. 3, pp. 70–78, feb 2016.

(cited on page 12.)

[62] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "MiniBox: a two-way sandbox for x86 native code," in *Proc. of the USENIX Annual Technical Conference (ATC)*, jun 2014, pp. 409–420.

(cited on pages 12, 82, and 112.)

[63] S. Shinde, D. L. Tien, S. Tople, and P. Saxena, “PANOPLY: Low-TCB Linux Applications with SGX Enclaves,” in *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

(cited on pages 12, 24, and 83.)

[64] T. Liu, C. Curtsinger, and E. D. Berger, “Dthreads: efficient deterministic multithreading,” in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011, p. 327.

(cited on page 14.)

[65] T. Bergan, O. Anderson, J. Devietti, L. Ceze, D. Grossman, T. Bergan, O. Anderson, J. Devietti, L. Ceze, D. Grossman, T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman, “CoreDet: a compiler and runtime system for deterministic multithreaded execution,” in *Proceedings of the 15th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 38, no. 1, 2010, p. 53.

(cited on page 14.)

[66] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant, “Parrot: a practical runtime for deterministic, stable, and reliable threads,” in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 388–405.

(cited on page 14.)

[67] B. Vavala, N. Neves, and P. Steenkiste, “Secure Identification of Actively Executed Code on a Generic Trusted Component,” in *Proceedings of the 46th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, jun 2016, pp. 419–430.

(cited on pages 16 and 50.)

[68] F. Tip, “A Survey of Program Slicing Techniques,” *Journal of Programming Languages*, vol. 3, 1995.

(cited on page 16.)

[69] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers, “Secure program partitioning,” *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 3, pp. 283–328, aug 2002.

(cited on pages 16 and 79.)

[70] T. Sander and C. F. Tschudin, “Towards mobile cryptography,” in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 1998, pp. 215–224.

(cited on page 22.)

[71] ——, “Protecting Mobile Agents Against Malicious Hosts,” *Mobile Agents and Security (LNCS)*, vol. 1419, no. 1998, pp. 44–60, jan 1998.
(cited on page 22.)

[72] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, and G. Tsudik, “Itinerant agents for mobile computing,” *IEEE Personal Communications*, vol. 2, no. 5, pp. 34–49, oct 1995.
(cited on page 22.)

[73] E. Shi, A. Perrig, and L. van Doorn, “BIND: A Fine-Grained Attestation Service for Secure Distributed Systems,” in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, may 2005, pp. 154–168.
(cited on pages 22, 52, and 78.)

[74] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri, “How low can you go?” in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 43, no. 3, mar 2008, p. 14.
(cited on pages 22 and 79.)

[75] D. Grawrock, *The Intel Safer Computing Initiative: Building Blocks for Trusted Computing*, ser. Engineer to Engineer Series. Intel Press, 2006.
(cited on pages 22 and 26.)

[76] ——, *Dynamics of a Trusted Platform: A Building Block Approach*. Intel Press, apr 2009.
(cited on pages 22, 52, and 62.)

[77] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, “AEGIS,” in *Proceedings of the 17th Annual International Conference on Supercomputing (ICS)*, 2003, pp. 160–171.
(cited on page 23.)

[78] E. Owusu, J. Guajardo, J. McCune, J. Newsome, A. Perrig, and A. Vasudevan, “OASIS,” in *Proceedings of the 2013 Conference on Computer & Communications Security (CCS)*, nov 2013, pp. 13–24.
(cited on pages 23, 52, and 78.)

[79] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, “TrustLite,” in *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, 2014, pp. 1–14.
(cited on page 23.)

[80] ARM Security Technology, “Building a Secure System using TrustZone Technology,” pp. 1–108.

(cited on page 24.)

[81] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten, "fTPM: A Software-only Implementation of a TPM Chip," in *Proceedings of the 25th USENIX Security Symposium*, 2016.

(cited on page 24.)

[82] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal Hardware Extensions for Strong Software Isolation," in *Proceedings of the 25th USENIX Security Symposium*, 2016.

(cited on pages 24 and 25.)

[83] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan : A Distributed Sandbox for Untrusted Computation on Secret Data," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 533–550.

(cited on pages 24 and 83.)

[84] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza, "SecureKeeper: Confidential ZooKeeper using Intel SGX," in *Proceedings of the International Middleware Conference (Middleware)*, 2016, pp. 1–13.

(cited on page 24.)

[85] S. Kim, J. Han, J. Ha, T. Kim, and D. Han, "Enhancing Security and Privacy of Tor's Ecosystem by using Trusted Execution Environments," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

(cited on page 24.)

[86] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An Oblivious and Encrypted Distributed Analytics Platform," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

(cited on page 24.)

[87] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer, "SGXBounds: Memory Safety for Shielded Execution," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2017.

(cited on page 24.)

[88] J. Behl, T. Distler, and R. Kapitza, "Hybrids on Steroids: SGX-Based High Performance BFT," in *Proceedings of the European Conference in Computer Systems (EuroSys)*, 2017.

(cited on page 24.)

[89] M. Orenbach, M. Minkin, P. Lifshits, and M. Silberstein, “Eleos: ExitLess OS services for SGX enclaves,” in *Proceedings of the European Conference in Computer Systems (EuroSys)*, 2017.
(cited on page 24.)

[90] V. Karande, E. Bauman, Z. Lin, and L. Khan, “SGX-Log,” in *Proceedings of the Asia Conference on Computer and Communications Security (CCS)*, 2017, pp. 19–30.
(cited on page 24.)

[91] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, “Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu,” in *Proceedings of the Asia Conference on Computer and Communications Security (CCS)*, 2017, pp. 7–18.
(cited on page 24.)

[92] S. Tamrakar, J. Liu, A. Paverd, J.-E. Ekberg, B. Pinkas, and N. Asokan, “The Circle Game,” in *Proceedings of the Asia Conference on Computer and Communications Security (CCS)*, 2017, pp. 31–44.
(cited on page 24.)

[93] J. Lind, C. Priebe, D. Muthukumaran, D. O’Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch, “Glamdring: Automatic Application Partitioning for Intel SGX,” in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.
(cited on page 24.)

[94] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta, “Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework,” in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, may 2013, pp. 430–444.
(cited on pages 25, 27, 30, 51, 53, 70, 107, 109, 112, and 119.)

[95] E. Brickell and J. Li, “Enhanced Privacy ID,” in *Proceedings of the ACM workshop on Privacy in electronic society (WPES)*, 2007, p. 21.
(cited on pages 29 and 30.)

[96] E. Brickell, J. Camenisch, and L. Chen, “Direct anonymous attestation,” in *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, 2004, p. 132.
(cited on page 30.)

[97] S. Checkoway and H. Shacham, “Iago attacks,” in *Proceedings of the 18th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 41, no. 1, mar 2013, p. 253.

(cited on pages 33, 83, and 85.)

[98] I. Anati and S. Gueron, “Innovative technology for cpu based attestation and sealing,” in *Proceedings of the 2nd Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

(cited on pages 40, 42, 62, 65, and 66.)

[99] R. Strackx and F. Piessens, “Ariadne: A Minimal Approach to State Continuity,” in *Proceedings of the 25th USENIX Security Symposium*, 2016, pp. 875–892.

(cited on pages 42 and 45.)

[100] C. J. Cremers, “The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols,” in *Proceedings of the 20th international conference on Computer Aided Verification (CAV)*, vol. 5123, Berlin, Heidelberg, jul 2008, pp. 414–418.

(cited on pages 51 and 71.)

[101] ——, “Unbounded verification, falsification, and characterization of security protocols by pattern refinement,” in *Proceedings of the 15th conference on Computer and Communications Security (CCS)*, oct 2008, p. 119.

(cited on pages 51 and 71.)

[102] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, “Design and implementation of a TCG-based integrity measurement architecture,” in *Proceedings of the 13th USENIX Security Symposium*, 2004, p. 16.

(cited on page 51.)

[103] B. Kauer, “OSLO: improving the security of trusted computing,” in *Proceedings of 16th USENIX Security Symposium*, aug 2007, p. 16.

(cited on page 51.)

[104] “Open Sourced Vulnerability Database,” *http://www.osvdb.org/*.

(cited on page 52.)

[105] “Exploit-DB,” *http://www.exploit-db.com/*.

(cited on page 52.)

[106] S. Bratus, N. D’Cunha, E. Sparks, and S. W. Smith, “TOCTOU, Traps, and Trusted Computing,” in *Proceedings of the 1st international conference on Trusted Computing and Trust in Information Technologies (TRUST)*, vol. 4968, 2008, pp. 14–32.

(cited on page 52.)

[107] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2nd Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013, pp. 1–1.
(cited on pages 52, 53, and 62.)

[108] Intel, "Software Guard Extensions," *https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf*.
(cited on page 53.)

[109] "SQLite in Android," *http://developer.android.com/reference/ android/ database/ sqlite/ SQLiteDatabase.html*.
(cited on page 70.)

[110] "SQLite in iCloud," *https://developer.apple.com/library/ios/ documentation/ DataManagement/ Conceptual/ UsingCoreData WithiCloudPG/ UsingSQLiteStoragewithi Cloud/UsingSQLiteStoragewithiCloud.html*.
(cited on page 70.)

[111] "SQLite Deployments," *http://sqlite.org/mostdeployed.html*.
(cited on page 70.)

[112] ZeroMQ, "http://zeromq.org/."
(cited on pages 71 and 134.)

[113] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill, "Ironclad apps: end-to-end security via automated full-system verification," in *Proc. of the 11th USENIX conference on Operating Systems Design and Implementation (OSDI)*, oct 2014, pp. 165–181.
(cited on page 72.)

[114] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in distributed systems: theory and practice," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 4, pp. 265–310, nov 1992.
(cited on page 78.)

[115] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. H. Jakubowski, "Oblivious Hashing: A Stealthy Software Integrity Verification Primitive," in *Proceedings of the 5th International Workshop on Information Hiding (IH)*, oct 2002, pp. 400–414.
(cited on page 78.)

[116] J.-E. Ekberg, N. Asokan, K. Kostiainen, and A. Rantala, "Scheduling execution of credentials in constrained secure environments," in *Proceedings of the Workshop on Scalable Trusted Computing (STC)*, 2008.

(cited on pages 78 and 79.)

[117] K. Kostiainen, J.-E. Ekberg, N. Asokan, and A. Rantala, "On-board credentials with open provisioning," in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security (ASIACCS)*, 2009, p. 104.

(cited on pages 78 and 79.)

[118] S. Bugiel and J.-E. Ekberg, "Implementing an application-specific credential platform using late-launched mobile trusted module," in *Proceedings of the 5th ACM workshop on Scalable trusted computing (STC)*, 2010, p. 21.

(cited on pages 78 and 79.)

[119] Trusted Computing Group, "MTM Specification v1.0 rev. 7.02," 2010.

(cited on page 78.)

[120] ——, "Mobile Trusted Module 2.0 Use Cases," 2011.

(cited on page 78.)

[121] D. Kilpatrick, "Privman: A library for partitioning applications," in *Proceedings of the USENIX Annual Technical Conference (Freenix track)*, 2003.

(cited on page 79.)

[122] M. Girkar and C. Polychronopoulos, "Partitioning programs for parallel execution," in *Proceedings of the 2nd Int. Conference on Supercomputing (ICS)*, 1988, pp. 216–229.

(cited on page 79.)

[123] E. Yardimci and M. Franz, "Mostly static program partitioning of binary executables," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 31, no. 5, pp. 1–46, jun 2009.

(cited on page 79.)

[124] S. Bajaj and R. Sion, "TrustedDB: A Trusted Hardware-Based Database with Privacy and Data Confidentiality," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 3, pp. 752–765, mar 2014.

(cited on pages 81 and 83.)

[125] T. T. A. Dinh, P. Saxena, E.-C. Chang, B. C. Ooi, and C. Zhang, "M 2 R: enabling stronger

privacy in mapreduce computation," in *Proceedings of the 24th USENIX Security Symposium (SEC)*, aug 2015, pp. 447–462.
(cited on pages 83 and 112.)

[126] B. Haubold and T. Wiehe, *Biological Sequences and the Exact String Matching Problem*. Birkhäuser Verlag, Basel (Switzerland), 2006.
(cited on pages 111 and 112.)

[127] J. C. Venter and Et-al., "The Sequence of the Human Genome," *Science*, vol. 291, no. 5507, pp. 1304–1351, 2001.
(cited on page 112.)

[128] W. Stallings and L. Brown, *Computer Security: Principles and Practice*, 3rd ed. Prentice Hall, 2014.
(cited on page 117.)

[129] A. S. Tanenbaum and M. Van Steen, *Distributed Systems: Principles and Paradigms*. Pearson Prentice Hall, 2002, vol. paperback.
(cited on page 117.)

[130] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," *Journal of the ACM (JACM)*, vol. 27, no. 2, pp. 228–234, 1980.
(cited on page 117.)

[131] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.
(cited on pages 117, 120, and 124.)

[132] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.
(cited on page 117.)

[133] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a Tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.
(cited on page 117.)

[134] P. A. Alsberg and J. D. Day, "A principle for resilient sharing of distributed resources," in *In Proc. of the 2nd International Conference on Software Engineering (ICSE)*, 1976, p. 562.
(cited on pages 117, 123, and 124.)

[135] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "Optimal Primary-Backup Pro-

tocols," in *Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG)*, 1992, p. 378.

(cited on page 117.)

[136] ——, "The primary-backup approach," in *Distributed systems (2nd Ed.)*, may 1993, pp. 199–216.

(cited on page 117.)

[137] N. Budhiraja and K. Marzullo, "Tradeoffs in implementing primary-backup protocols," in *Proceedings of the 7th International Parallel and Distributed Processing Symposium (IPDPS)*, 1995, pp. 280–288.

(cited on page 117.)

[138] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 398–461, nov 2002.

(cited on pages 117, 118, and 136.)

[139] M. Correia, N. Neves, and P. E. Veríssimo, "How to Tolerate Half Less One Byzantine Nodes in Practical Distributed Systems," in *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS)*, 2004, pp. 174–183.

(cited on pages 118, 119, and 125.)

[140] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, "CheapBFT: resource-efficient byzantine fault tolerance," in *Proceedings of the 7th European Conference on Computer Systems (EuroSys)*, 2012, p. 295.

(cited on pages 118, 124, 125, and 136.)

[141] G. S. Veronese, M. Correia, A. Bessani, L. C. Lung, and P. E. Veríssimo, "Efficient Byzantine Fault-Tolerance," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 16–30, jan 2013.

(cited on pages 118 and 125.)

[142] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet, "ZZ and the art of practical BFT execution," in *Proceedings of the 6th Conference on Computer Systems (EuroSys)*, 2011, pp. 123–138.

(cited on page 118.)

[143] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making Byzantine fault tolerant systems tolerate Byzantine faults," *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (OSDI)*, pp. 153–168, 2009.

(cited on page 118.)

[144] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 BFT protocols," in *Proceedings of the 5th European conference on Computer systems (EuroSys)*, 2010, p. 363.
(cited on page 118.)

[145] R. Garcia, R. Rodrigues, and N. Preguiça, "Efficient middleware for byzantine fault tolerant database replication," in *Proceedings of the 6th European conference on Computer systems (EuroSys)*, 2011, pp. 107–122.
(cited on page 118.)

[146] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "HQ replication: a hybrid quorum protocol for byzantine fault tolerance," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006, pp. 177–190.
(cited on page 118.)

[147] J.-P. Martin and L. Alvisi, "Fast Byzantine Consensus," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, jun 2005, pp. 402–411.
(cited on page 118.)

[148] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. L. Wong, "Zyzzyva: Speculative Byzantine Fault Tolerance," in *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP)*, vol. 41, no. 6, 2007, pp. 45–58.
(cited on page 118.)

[149] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for byzantine fault tolerant services," in *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, vol. 37, no. 5, 2003.
(cited on pages 118 and 124.)

[150] T. Distler, I. Popov, and W. Schröder-Preikschat, "SPARE: Replicas on Hold." in *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, 2011.
(cited on page 118.)

[151] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested append-only memory: making adversaries stick to their word," in *Proceedings of 21st Symposium on Operating Systems Principles (SOSP)*, vol. 41, no. 6, 2007, p. 189.
(cited on pages 118, 124, and 125.)

[152] I. Abraham, M. K. Aguilera, and D. Malkhi, "Fast Asynchronous Consensus with Optimal Resilience," in *Proceedings of the 24th Conference on Distributed Computing (DISC)*, 2010, pp. 4–19.

(cited on pages 118 and 125.)

[153] J. Hendricks, S. Sinnamohideen, G. R. Ganger, and M. K. Reiter, "Zzyzx: Scalable fault tolerance through Byzantine locking," in *IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, jun 2010, pp. 363–372.
(cited on page 118.)

[154] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, "All about Eve: execute-verify replication for multi-core servers," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI)*, oct 2012, pp. 237–250.
(cited on pages 118, 125, and 137.)

[155] P.-L. Aublin, S. B. Mokhtar, and V. Quema, "RBFT: Redundant Byzantine Fault Tolerance," in *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS)*, 2013, pp. 297–306.
(cited on pages 118 and 133.)

[156] J. G. Slember and P. Narasimhan, "Static analysis meets distributed fault-tolerance: enabling state-machine replication with nondeterminism," in *Proceedings of the 2nd Conference on Hot topics in System Dependability (HotDep)*, nov 2006, p. 2.
(cited on page 118.)

[157] R. van Renesse and R. Guerraoui, "Replication techniques for availability," in *Replication*. Springer-Verlag, jan 2010, pp. 19–40.
(cited on page 118.)

[158] J. Antunes and N. Neves, "DiveInto: Supporting Diversity in Intrusion-Tolerant Systems," in *Proceedings of the 30th Symposium on Reliable Distributed Systems (SRDS)*, 2011, pp. 137–146.
(cited on page 118.)

[159] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP)*, vol. 37, no. 5, dec 2003, p. 29.
(cited on page 118.)

[160] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou, "Boxwood: abstractions as the foundation for storage infrastructure," in *Proceedings of the 6th Symposium on Opearting Systems Design & Implementation (OSDI)*, dec 2004, p. 8.
(cited on page 118.)

[161] J. Maccormick, C. A. Thekkath, M. Jager, K. Roomp, L. Zhou, and R. Peterson, “Niobe: a Practical Replication Protocol,” *Journal ACM Transactions on Storage (TOS)*, vol. 3, no. 4, pp. 1–43, feb 2008.

(cited on page 118.)

[162] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, “Remus: high availability via asynchronous virtual machine replication,” in *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI)*, apr 2008, pp. 161–174.

(cited on page 118.)

[163] J. R. Lorch, A. Baumann, L. Glendenning, D. T. Meyer, and A. Warfield, “Tardigrade: leveraging lightweight virtual machines to easily and efficiently construct fault-tolerant services,” in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2015, pp. 575–588.

(cited on page 118.)

[164] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, “ZooKeeper: wait-free coordination for internet-scale systems,” in *Proceedings of the USENIX Annual Technical Conference (ATC)*, jun 2010, p. 11.

(cited on page 118.)

[165] M. Correia, N. Neves, L. C. Lung, and P. E. Veríssimo, “Low complexity Byzantine-resilient consensus,” *Distributed Computing*, vol. 17, no. 3, p. 237, 2005.

(cited on pages 119 and 125.)

[166] P. E. Veríssimo, “Travelling through wormholes,” *ACM SIGACT News*, vol. 37, no. 1, p. 66, mar 2006.

(cited on pages 119 and 125.)

[167] J. Sousa, E. Alchieri, and A. Bessani, “State Machine Replication for the Masses with BFT-SMaRt,” in *Proceedings of the IEEE Conference on Dependable Systems & Networks (DSN)*, 2014, pp. 355–362.

(cited on pages 119, 124, and 133.)

[168] Y. Amir, B. Coan, J. Kirsch, and J. Lane, “Prime: Byzantine Replication under Attack,” *IEEE Transactions on Dependable and Secure Computing (TDSC)*, vol. 8, no. 4, pp. 564–577, jul 2011.

(cited on pages 119, 133, and 136.)

[169] X. Defago, A. Schiper, and N. Sergent, “Semi-passive replication,” in *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, 1998, pp. 43–50.

(cited on page 123.)

[170] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro, “OS diversity for intrusion tolerance: Myth or reality?” in *Proceedings of the 41st International Conference on Dependable Systems & Networks (DSN)*, jun 2011, pp. 383–394.

(cited on page 123.)

[171] ——, “Analysis of operating system diversity for intrusion tolerance,” *Software: Practice and Experience*, vol. 44, no. 6, pp. 735–770, jun 2014.

(cited on page 123.)

[172] IBM, “System z10,” *http://www.redbooks.ibm.com/redbooks/pdfs/ /sg247516.pdf*.

(cited on page 124.)

[173] C. Dwork, N. Lynch, and L. Stockmeyer, “Consensus in the presence of partial synchrony,” *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.

(cited on page 124.)

[174] D. Dolev, C. Dwork, and L. Stockmeyer, “On the minimal synchronism needed for distributed consensus,” *Journal of the ACM*, vol. 34, no. 1, pp. 77–97, 1987.

(cited on page 124.)

[175] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, “Zyzzyva: Speculative Byzantine Fault Tolerance,” *ACM Transactions on Computer Systems (TOCS)*, vol. 27, no. 4, p. 7, dec 2009.

(cited on page 124.)

[176] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, “The Next 700 BFT Protocols,” *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 4, pp. 1–45, jan 2015.

(cited on pages 124 and 133.)

[177] M. J. Fischer, N. Lynch, and M. S. Paterson, “Impossibility of distributed consensus with one faulty process,” *Journal of the ACM (JACM)*, vol. 32, no. 2, 1985.

(cited on page 124.)

[178] M. Castro and B. Liskov, “Practical byzantine fault tolerance and proactive recovery,” *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 398–461, 2002.

(cited on page 124.)

[179] D. Dolev and R. Reischuk, “Bounds on information exchange for Byzantine agreement,” *Journal of the ACM (JACM)*, vol. 32, no. 1, p. 191, 1985.

(cited on page 124.)

[180] R. Rodrigues, M. Castro, and B. Liskov, "BASE: using abstraction to improve fault tolerance," in *Proceedings of the eighteenth Symposium on Operating Systems Principles (SOSP)*, vol. 35, no. 5, 2001, p. 15.
(cited on pages 125 and 137.)

[181] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda, "TrInc: small trusted hardware for large distributed systems," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009, pp. 1–14.
(cited on page 125.)

[182] M. Correia, G. S. Veronese, and L. C. Lung, "Asynchronous Byzantine consensus with 2f+1 processes," in *Proceedings of the ACM Symposium on Applied Computing (SAC)*, mar 2010, p. 475.
(cited on page 125.)

[183] F. Sheldon, D. Fetzer, D. Manz, J. Huang, S. Goose, T. Morris, J. Dang, J. Kirsch, and D. Wei, "Intrinsically resilient energy control systems," in *Proceedings of the 8th Annual Cyber Security and Information Intelligence Research Workshop (CSIIRW)*, 2013, p. 1.
(cited on page 133.)

[184] J. Behl, T. Distler, and R. Kapitza, "Consensus-Oriented Parallelization," in *Proceedings of the 16th Annual Conference on Middleware*, 2015, pp. 173–184.
(cited on page 133.)

[185] T. Distler, C. Bahn, A. Bessani, F. Fischer, and F. Junqueira, "Extensible distributed coordination," in *Proceedings of the 10th European Conference on Computer Systems (Eurosys)*, 2015.
(cited on page 133.)

[186] R. Guerraoui, A.-M. Kermarrec, M. Pavlovic, and D.-A. Seredinschi, "Atum: Scalable Group Communication Using Volatile Groups," in *Proceedings of the 17th International Conference Middleware*, 2016, pp. 1–14.
(cited on page 133.)

[187] A.-R. Sadeghi and C. Stüble, "Property-based attestation for computing platforms," in *Proceedings of the workshop on New Security Paradigms (NSPW)*, sep 2004, p. 67.
(cited on page 144.)

[188] SQLite. WWW.SQLITE.ORG

(cited on pages 17, 51, 70, 111, 112, 119, and 134.)

[189] Intel. Intel Software Guard Extensions. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf

(cited on pages 4, 7, 24, 25, 28, 30, 43, 77, 102, 103, and 140.)

[190] Microsoft. Minimum hardware requirements for Windows 10. https://msdn.microsoft.com/en-us/library/windows/hardware/dn915086

(cited on page 25.)

[191] Intel. IntelÍ SGX SDK for Windows* User Guide. https://software.intel.com/sites/default/files/managed/b4/cf/Intel-SGX-SDK-Developer-Reference-for-Windows-OS.pdf

(cited on pages 43 and 44.)

[192] A. Baumann, M. Peinado, G. Hunt, K. Zmudzinski, C. V. Rozas, M. Hoekstra. ”Secure execution of unmodified applications on an untrusted host”. Poster/Work-in-Progress. SOSP, 2013. http://research.microsoft.com/pubs/204758/sosp13-abstract.pdf

(cited on page 13.)

[193] IBM. IBM Systems Cryptographic Hardware Products. http://www-03.ibm.com/security/cryptocards/

(cited on page 6.)

[194] Intel Software Guard Extensions: EPID Provisioning and Attestation Services. https://software.intel.com/sites/default/files/managed/ac/40/2016%20WW10%20sgx%20provisioning%20and%20attesatation%20final.pdf

(cited on pages 6, 29, 30, 38, 39, and 91.)

[195] David A. Wheeler. SLOCCount. http://www.dwheeler.com/sloccount/sloccount.html

(cited on pages 108 and 135.)

[196] Cloud Security Alliance. The Treacherous 12 – Cloud Computing Top Threats in 2016. https://downloads.cloudsecurityalliance.org/assets/research/top-threats/Treacherous-12_Cloud-Computing_Top-Threats.pdf

(cited on page 2.)

[197] Kernel Statistics. http://linuxcounter.net/statistics/kernel

(cited on page 12.)

[198] GNU C Library. https://www.openhub.net/p/glibc/analyses/latest/languages_summary

(cited on page 12.)

[199] AMD. Secure Memory Encryption. http://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

(cited on pages 6, 7, 24, and 25.)

[200] AMD. Secure Encrypted Virtualization. http://support.amd.com/TechDocs/55766_SEV-KM%20API_Spec.pdf

(cited on pages 4, 6, 7, 24, 25, and 30.)

[201] AMD. AMD64 Architecture Programmer's Manual Volume 2: System Programming. http://support.amd.com/TechDocs/24593.pdf

(cited on page 7.)

[202] Dell Optiplex 7040. http://www.dell.com/support/manuals/pt/pt/ptbsdt1/optiplex-7040-desktop/Opti7040_SFF_OM/System-Setup-options?guid=GUID-50D00E06-502E-4575-83EE-8682F97BF667

(cited on page 85.)

[203] Cisco. Cisco Cloud Services Platform 2100 Remote Command Execution Vulnerability (CVE-2016-6374). https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20160921-csp2100-2

(cited on page 2.)

[204] Amazon. Amazon Linux AMI Security Advisory: ALAS-2016-653. https://alas.aws.amazon.com/ALAS-2016-653.html

(cited on page 2.)

[205] Cisco. Cisco Cloud Services Platform 2100 Command Injection Vulnerability (CVE-2016-6373). https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20160921-csp2100-1

(cited on page 2.)

[206] Rackspace. QEMU ”VENOM” Vulnerability (CVE-2015-3456). https://community.rackspace.com/general/f/53/t/5187

(cited on page 2.)

[207] Illumina. Sequencing Systems. http://illumina.com/systems/sequencing.html

(cited on page 112.)

[208] DNAnexus Sequence Read Archive. Homo Sapiens SRR622458-NA12891. HTTP://SRA.DNANEXUS.COM/RUNS/SRR622458

(cited on page 112.)

[209] USA Today. Facebook has a billion users in a single day, says Mark Zuckerberg. HTTP://WWW.BBC.COM/NEWS/WORLD-US-CANADA-34082393

(cited on page 1.)

[210] BBC. WhatsApp reaches a billion monthly users. HTTP://WWW.BBC.COM/NEWS/TECHNOLOGY-35459812

(cited on page 1.)

[211] Visa. Visa acceptance for retailers. HTTPS://USA.VISA.COM/RUN-YOUR-BUSINESS/SMALL-BUSINESS-TOOLS/RETAIL.HTML

(cited on page 1.)

[212] Amazon. All Customer Success Stories. HTTPS://AWS.AMAZON.COM/SOLUTIONS/CASE-STUDIES/ALL

(cited on page 1.)

[213] Amazon. Illumina Case Study. HTTPS://WWW.RESILIENTSYSTEMS.COM/CYBER-RESILIENCE-KNOWLEDGE-CENTER/INCIDENT-RESPONSE-BLOG/SECURITY-BUSINESS-FLEXIBILITY

(cited on page 1.)

[214] Amazon. AWS Global Infrastructure. HTTPS://AWS.AMAZON.COM/ABOUT-AWS/GLOBAL-INFRASTRUCTURE

(cited on page 2.)

[215] Microsoft. Azure regions. HTTPS://AZURE.MICROSOFT.COM/EN-US/REGIONS

(cited on page 2.)

[216] Google. Google Cloud Platform Locations. HTTPS://CLOUD.GOOGLE.COM/ABOUT/LOCATIONS/

(cited on page 2.)

[217] US Department of Defense. Orange Book, DoD 5200.28-STD. Trusted Computer System Evaluation Criteria (TCSEC). HTTPS://UPLOAD.WIKIMEDIA.ORG/WIKIPEDIA/COMMONS/A/AA/TRUSTED_COMPUTER_SYSTEM_EVALUATION_CRITERIA_DOD_5200.28-STD.PDF

(cited on page 12.)

[218] Bruno Vavala, Nuno Neves, Peter Steenkiste. Secure Tera-scale Data Crunching with a Small TCB. (submitted for publication)

(not cited.)

[219] Intel. Intel Software Guard Extensions Remote Attestation End-to-End Example. HTTPS://SOFTWARE.INTEL.COM/EN-US/ARTICLES/INTEL-SOFTWARE-GUARD-EXTENSIONS-REMOTE-ATTESTATION-END-TO-END-EXAMPLE

(cited on pages 39 and 46.)

[220] Intel. Intel Attestation Service (IAS). HTTPS://AS.SGX.TRUSTEDSERVICES.INTEL.COM:443

(cited on pages 39 and 106.)

[221] Intel. Intel Attestation Service API. HTTPS://SOFTWARE.INTEL.COM/SITES/DEFAULT/FILES/MANAGED/3D/C8/IAS_1_0_API_SPEC_1_1_FINAL.PDF

(cited on pages 39, 46, and 106.)

[222] C. Tsai, D. Porter. "Graphene / Graphene-SGX Library OS - a library OS for Linux multi-process applications, with Intel SGX support". HTTPS://GITHUB.COM/OSCARLAB/GRAPHENE

(cited on pages 8 and 83.)

[223] IBM. IBM Financial Transaction Manager. HTTPS://WWW.IBM.COM/US-EN/MARKETPLACE/FINANCIAL-TRANSACTION-SOFTWARE

(cited on page 1.)

[224] Oracle. Oracle Financials Cloud. HTTPS://CLOUD.ORACLE.COM/EN_US/FINANCIALS-CLOUD

(cited on page 1.)

[225] Intel. Intel Skylake Products. HTTP://ARK.INTEL.COM/PRODUCTS/CODENAME/37572/SKYLAKE

(cited on page 27.)

[226] Intel. Intel Kaby Lake Products. HTTPS://ARK.INTEL.COM/PRODUCTS/CODENAME/82879/KABY-LAKE

(cited on page 27.)