Technische
Universität
Braunschweig

Systems Support for Emerging Memory Technologies

Von der
Carl-Friedrich-Gauß-Fakultät
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines

Doktoringenieurs (Dr.-Ing.)

genehmigte Dissertation

von
Vasily Sartakov
geboren am **25.09.1985**
in **Tscheljabinsk**

Eingereicht am:	**21.12.2018**
Disputation am:	**03.05.2019**
1. Referent:	**Prof. Dr. Rüdiger Kapitza**
2. Referent:	**Prof. Dr. Christof Fetzer**

(2019)

Kurzfassung

Vertrauenswürdigkeit und Skalierbarkeit sind die beiden maßgeblichen Faktoren, die die Verbreitung von Clouds behindern. Die Möglichkeit privilegierter Zugriffe auf Kundendaten durch einen Cloudanbieter schränkt die Nutzung von Clouds bei der Verarbeitung von sicherheitskritischen und vertraulichen Informationen ein. Clouddienste mit niedriger Latenz erfordern die Durchführungen von Berechnungen im Hauptspeicher und sind daher an Charakteristika von Dynamic RAM (DRAM) wie Kapazität, Dichte, Energieverbrauch und andere Aspekte gebunden.

Zwei technologische Bereiche befassen sich mit diesen Faktoren: Etablierte Server Plattformen wie Intel Software Guard eXtensions (SGX) und AMD Secure Encrypted Virtualisation (SEV) stellen Erweiterungen für vertrauenswürdige Ausführung in nicht vertrauenswürdigen Umgebungen bereit. Verschiedene Technologien von nicht flüchtigem Speicher bieten bessere Kapazität und Speicherdichte verglichen mit DRAM, und können daher in Zukunft als Alternative zu DRAM herangezogen werden. Jedoch benötigen diese Technologien und Erweiterungen neuartige Ansätze und Systemunterstützung bei der Programmierung, da diese der Systemarchitektur neue Funktionalität hinzufügen: Systemkomponenten (Intel SGX) und Persistenz (nicht-flüchtiger Speicher).

Diese Dissertation widmet sich der Programmierung und den Architekturaspekten von persistenten und vertrauenswürdigen Systemen. Für vertrauenswürdige Systeme wurde eine detaillierte Analyse der neuen Architekturerweiterungen durchgeführt. Außerdem wurden das neuartige "*E*Actors" Framework und die "STANlite" Datenbank entwickelt, um die neuen Möglichkeiten von vertrauenswürdiger Ausführung effektiv zu nutzen. Darüber hinaus wurde für persistente Systeme eine detaillierte Analyse zukünftiger Speichertechnologien, deren Merkmale und mögliche Auswirkungen auf die Systemarchitektur durchgeführt. Ferner wurde das neue *Hypervisor-basierte* Persistenzmodell entwickelt und mittels NV-Hypervisor ausgewertet, welches transparente Persistenz für alte und proprietäre Software, sowie Virtualisierung von persistentem Speicher ermöglicht.

Abstract

Trust and scalability are the two significant factors which impede the dissemination of clouds. The possibility of privileged access to customer data by a cloud provider limits the usage of clouds for processing security-sensitive data. Low latency cloud services rely on in-memory computations, and thus, are limited by several characteristics of DRAM such as capacity, density, energy consumption, for example.

Two technological areas address these factors. Mainstream server platforms, such as Intel Software Guard eXtensions (SGX) and AMD Secure Encrypted Virtualisation (SEV) offer extensions for trusted execution in untrusted environments. Various technologies of Non-Volatile RAM (NV-RAM) have better capacity and density compared to DRAM and thus can be considered as DRAM alternatives in the future. However, these technologies and extensions require new programming approaches and system support since they add features to the system architecture: new system components (Intel SGX) and data persistence (NV-RAM).

This thesis is devoted to the programming and architectural aspects of persistent and trusted systems. For trusted systems, an in-depth analysis of new architectural extensions was performed. A novel framework named *E*Actors and a database engine named STANlite were developed to effectively use the capabilities of trusted execution.

For persistent systems, an in-depth analysis of prospective memory technologies, their features and the possible impact on system architecture was performed. A new persistence model, called the *hypervisor-based* model of persistence, was developed and evaluated by the NV-Hypervisor. This offers transparent persistence for legacy and proprietary software, and supports virtualisation of persistent memory.

Table of Contents

List of Abbreviations

AEX Asynchronous Enclave Exit ... 18

API Application Programming Interface ... 26

BBRAM Battery-backed RAM ... 95

CERB Client Encrypted Request Buffer ... 78

CPRB Client Plain text Request Buffer ... 78

DAX Direct Access ... 102

DB Database ... 66

CONF CONFiguration section ... 115

DMA Direct Memory Access ... 11

ECC Error-correcting code ... 101

EDL Enclave Description Language ... 19

ELRANGE Enclave Linear Address Range ... 13

EOS *E*actors Object Store ... 31

EPCM Enclave Page Cache Map ... 12

EPC Enclave Page Cache ... 12

FeRAM Ferroelectric RAM ... 95

HDD Hard Disk Drive ... 93

HLE Hardware Lock Elision ... 31

IaaS Infrastructure-as-a-Service ... 9

JID JabberID ... 48

KVS Key-Value Store ... 38

LRU Least Recently Used ... 72

MAC Message Authentication Code ... 15

MRAM Magnetoresistive RAM ... 95

MRR Memory Residence Ratio ... 118

MTJ Magnetic Tunnel Junction ... 98

NVDIMM Non-Volatile Dual In-line Memory Module ... 95

NV-RAM	Non-Volatile RAM	94
O2M	One-to-Many	46
O2O	One-to-One	46
pVM	Persistent Virtual Machine	110
PaaS	Platform-as-a-Service	5
PC-RAM	Phase-Change RAM	95
PLC	Private List of Clients	48
POD	Power Outage Detector	95
PPT	Persistent Page Table	114
PRM	Processor Reserved Memory	12
PTE	Page Table Entry	11
RDMA	Remote Direct Memory Access	68
RPC	Remote Procedure Call	9
SaaS	Software-as-a-Service	5
SDK	Software Development Kit	15
SECS	SGX Enclave Control Structure	12
SERB	Server Encrypted Request Buffer	78
SEV	Secure Encrypted Virtualisation	9
SGX	Software Guard eXtensions	9
SLOC	Source Lines of Code	82
SMC	secure multi-party computation	43
SME	Secure Memory Encryption	10
SPRB	Server Plain text Request Buffer	79
SSA	State Save Area	13
SSL	Secure Sockets Layer	57
STT-MRAM	Spin Transfer Torque Magnetoresistive RAM	98
TCB	Trusted Computing Base	10
TCS	Thread Control Structure	13
ToC	Table of Contents	115
TPM	Trusted Platform Module	10
TpS	Transactions per Second	89
VME	Virtual Memory Engine	66
VPM	Virtual Persistent Memory	112

List of Figures

List of Tables

1. Introduction

Outsourcing of data processing applications to cloud infrastructures has become best practice. Clouds offer lower costs and robust scalability of services, which are essential for both end users and enterprises. The growing market of public cloud services supports this statement: the market has grown approximately 18% each year since 2009 and reached $153B in 2017 [3]. According to Gartner, this trend will continue: by the year 2021, the market will double [4]. While analysts forecast further growth, they also pay attention to market challenges such as *trust* [5] and *scalability* [6].

Trust is the fundamental factor of cloud computing. A cloud user expects that their data is protected, and neither other users nor the staff of a cloud provider can access it. Unauthorised access to private data[1], data breaches[2], long-lived critical software vulnerabilities[3], and, finally, hardware vulnerabilities such as Spectre [9] and Meltdown [10] – these and many other incidents demonstrate that expectations are not met.

Scalability is another important factor of cloud computing. Growing demands in cloud services are satisfied by the growing performance of CPUs, network bandwidth, storage sizes, and main memory capacity. The latter plays a crucial role because main memory has become a driver in the growth of cloud computing. Low-latency cloud services rely on in-memory data processing, i.e. a form of data processing which eliminates slow I/O of secondary storage by the storing and processing data inside main memory only [11]. Growing loads of cloud applications and scalability demands require providers to use more and more main memory for data processing. However, several factors impede the satisfaction of this need:

---

[1] In 2014 Facebook was suspected of using private data for commercial purposes [7]

[2] In 2014, attackers entered into a corporate network of eBay using credentials of ex-employees. As a result, information about 145 millions of eBay users was leaked [8]

[3] CVE-2018-8781 – eight-year-old critical vulnerability

- **Technological limitations**: Large volumes of main memory are unavailable due to the physical sizes of modules. In contrast to NAND memory technologies, which are used in Solid-State Drives (SSDs), Dynamic RAM (DRAM) cells require several times more space [12]
- **Growing energy consumption**: DRAM modules are built in the form of arrays of capacitors. These capacitors require constant powering to hold charges, whose need for electric current grows with the decreasing of cell sizes [13] and increasing number of modules.
- **Data loss risks**: Cloud infrastructure loses volatile data in the case of a power failure. Increasing in-memory data volumes results in an increase of recovery time and costs of power outages, the value of which constantly grows [14].

Two memory-related research areas address these challenges. The first one, devoted to *memory encryption and trusted execution*, is considered in the next section. The second one, dealing with *non-volatile memory technologies and persistent systems*, is considered below.

Memory encryption and trusted execution

Technologies of main memory encryption play a central role in trusted execution. Originally, they were used to prevent attacks on memory content. In-memory data can be retrieved directly by a cold-boot attack [15], or remotely by a Direct Memory Access (DMA) attack [16]. In both these cases, an attacker obtains a memory image, and encrypting of the memory prevents data leakage [17]. Nowadays, modern mainstream server platforms include extensions for trusted execution with memory encryption in their bases.

AMD Secure Encrypted Virtualisation (SEV) [18] enables trusted execution in the form of encrypted Virtual Machines (VMs), isolated from all privileged software and peripheral devices. Physical memory pages of these VMs are encrypted, and processes of encryption/decryption take place inside a memory controller without exposing cryptographic keys. Intel Software Guard eXtensions (SGX) [19] also uses memory encryption to enable trusted execution, but provides this in the form of new system components called *enclaves*.

Enclaves are isolated regions of code and data inside user-space programs. Only code located inside an enclave can access data of the same enclave [19]. As a consequence, in contrast to other system components, some restrictions apply to enclaves. Firstly, enclaved programs, i.e. programs executing inside enclaves, are self-contained and can not have any dependencies beyond the borders of their own enclave [20]. Secondly, the programs cannot issue system calls. Communication with the kernel can be performed via costly transition calls, named ECALLs and OCALLs, based on the SGX-specific instructions `EENTER` and `EEXIT`, respectively. Transitions cost at least 8000 CPU cycles, which is approximately 50 times slower than that of ordinary system calls [21]. Therefore, all kinds of operations which require the involvement of the kernel, in particular synchronisation mechanisms, become very costly. Thirdly, the use of enclaves causes heavyweight paging if the total size of memory used by all enclaves reaches the border in approximately 92 MiB [22].

Subject to these limitations, several programming approaches and frameworks for SGX-based trusted execution were proposed. Firstly, Intel provides SGX Software Development Kit (SDK), which enables integration of enclaves and programs in a Remote Procedure Call (RPC) manner [23]. A developer can implement functions which need to be executed inside enclaves, and an RPC interface for interaction of untrusted software with the enclaved functions. This approach is suitable for development of applications where enclaved software augments the untrusted part. SecureKeeper [24] is an example of a system which successfully applies this approach to secure data processing in untrusted cloud setting. While SGX SDK provides necessary primitives to program enclaves, it does not mitigate the mentioned restrictions of enclaves.

Secondly, enclaves can be considered as a secure environment for execution of legacy programs. For example, a database can provide confidential computations while being executed inside an enclave. In this scheme, untrusted software provides mechanisms for interaction with hardware, while the enclaved software performs the whole computations and data processing. To port existing services into this restricted environment, several frameworks, such as SCONE [25], Haven [26], and Graphene-SGX [27], were proposed. Some of them, like SCONE, provide an asynchronous call interface, which eliminates the transition costs of ECALLs.

While these frameworks mitigate some restrictions of enclave programming, they have a disadvantage. They offer single-enclave monolithic solutions in which all components of enclaved software share the same enclave. This approach increases the size of the Trusted Computing Base (TCB), which in turn increases the attack surface. Meanwhile, a single process can host multiple enclaves, and thus, an application can have a partitioned, low-TCB, multi-enclave design. The frameworks do not consider such programming models nor provide the corresponding system support, such as mechanisms for enclave-to-enclave communication.

Based on this, one can ask the research question: **What is a better programming model and system support for SGX enclaves?**

Non-volatile memory and persistent systems

Technological limitations, growing energy consumption and data loss risks can be mitigated by emerging memory technologies such as Phase-Change RAM (PC-RAM), Magnetoresistive RAM (MRAM) and others. Compared to DRAM, these technologies have two significant advantages: better capacity and density, and *persistence*. The first advantage can significantly increase the volume of available main memory and make them comparable in size with SSD [28, 29]. The second advantage resolves an issue of growing energy consumption and data loss risks since persistent memory does not require the use of an external source of power to store data [30].

Technologies of persistent memory cannot replace DRAM ”as is” because of several reasons. Firstly, persistence of the main memory requires revising the whole software stack. For instance, a program life-cycle, which includes steps of creation of an execution context in main memory, data copying from secondary storage to primary, the execution itself and memory deallocation after termination, is based on the assumption that main memory is volatile. However, if the main memory becomes persistent, the need for the secondary storage disappears, and persistent in-memory programs require a new life-cycle [31]. Secondly, technologies of persistent memory require special system support since they have their own features and characteristics, like asymmetry in read/write latencies, low durability and more [32, 12, 33, 34]. Fault models of Non-Volatile RAM (NV-RAM) technologies differ [35] from the fault models of DRAM and SSD and thus, system support requires developments of new approaches.

Various research works considered *models of persistence*, i.e. generalised architectures and programming models of persistent systems. Before developments of non-volatile memory technologies, projects like KeyKOS [36], EROS [37], and Coyotos [38] implemented checkpoints-based persistence. In this approach, an operating system duplicates memory pages of a target object on the secondary storage, and transparently recovers them from the storage after a power failure. Newton OS, which was used in the first PDAs of Apple, applied a different model of persistence. The Newton platform used battery-backed main memory which made all data stored inside it persistent. Newton OS used this persistent storage as object storage, and provided a unified access layer to distribute over the main memory and optional flash device objects [39]. Later, three additional persistence models were introduced for NV-RAM-based platforms.

The first one, the library and language-based model [40, 41], provides persistence of user-space objects. These objects can be created and used inside persistent memory using a special memory allocator, integrated into the programming environment. The second one, the process-based model [42], provides more coarse-grained persistence by allocating processes inside persistent memory. All objects produced by this kind of process are persistent. Finally, the system-wide model [43] generalises ideas of persistent processes to a whole system. In this design, all components of a system are located inside persistent memory and become persistent per se.

The significant disadvantage of these models, which limits their application for cloud computing, is a need for modifications of existing systems to utilise the support of NV-RAM. While it can be performed for Software-as-a-Service (SaaS) and Platform-as-a-Service (PaaS) clouds, where a service provider controls most components of a software stack, it is unlikely to be performed for Infrastructure-as-a-Service (IaaS) clouds, where a service provider manages only fundamental resources such as CPU cores, memory volumes and storage capacity [44]. In other words, in IaaS clouds, persistent memory support should be integrated into the guest software, which may be impossible since such clouds are often used for virtualisation of proprietary and legacy software.

It is with this in mind that one can ask the research question: **What is a better model of persistence and system support for persistent memory?**

1.1 Main Contributions

The thesis presents three main contributions: two related to programming models (C1) and system support of SGX-based trusted execution (C2), and one related to programming models and system support of persistent systems (C3):

C1: *E*Actors: actor-based framework for trusted execution

*E*Actors is a programming framework tailored for SGX. It features lightweight fine-grained parallelism based on the concept of actors. This approach avoids the use of costly synchronisation primitives, while lightweight nature of the actors significantly decreases the TCB size. Flexible and seamless design enables reconfiguration of services in accordance with security and performance demands. Multiple use cases were considered during evaluation, including schemes with multiple interacting enclaves.

C2: STANlite: a database engine for secure data processing at rack scale level

STANlite is an in-memory database engine for SGX-enabled secure data processing. It addresses challenges of enclave's paging and effective interaction of enclaved software with devices. The engine features small TCB and performs memory virtualisation without involvement of hardware-based paging. It demonstrated up to 2.44× performance increase compared to a vanilla SGX-based baseline.

C3: Hypervisor-based persistence and NV-Hypervisor

The conception of *hypervisor-based* persistence was introduced and evaluated by NV-Hypervisor. NV-Hypervisor provides transparent persistence for legacy and proprietary applications executed inside VMs. The hypervisor supports virtualisation of persistent memory and was evaluated on top of battery-backed NV-RAM. The evaluation demonstrated the significant decrease of a performance recovery time for a persistent database and low overhead impact of the virtual memory engine.

1.2 Related publications

- Vasily A. Sartakov, Stefan Brenner, Sonia Ben Mokhtar, Sara Bouchenak, Gaël Thomas and Rüdiger Kapitza: *E*Actors: Fast and flexible trusted computing using SGX, Proceedings of the 19th International Middleware Conference. ACM, 2018
- Vasily A. Sartakov, Nico Weichbrodt, Sebastian Krieter, Thomas Leich and Rüdiger Kapitza: STANlite - a database engine for secure data processing at rack-scale level, in IEEE International Conference on Cloud Engineering (IC2E), Orlando, Florida, U.S.A., 2018
- Vasily A. Sartakov, Arthur Martens and Rüdiger Kapitza: Temporality a NVRAM-based virtualization platform, in 34th IEEE Symposium on Reliable Distributed Systems (SRDS '15), Montreal, Canada, 2015
- Vasily A. Sartakov and Rüdiger Kapitza: NV-Hypervisor: Hypervisor-based Persistence for Virtual Machines, in Workshop on Dependability of Clouds, Data Centers and Virtual Machine Technology (DCDV 2014), Atlanta, 2014

1.3 Organisation

The thesis has two main parts. The first part (chapter 2) of the thesis is devoted to main memory encryption and trusted execution. Section 2.2 presents an actor-based framework and programming model for Intel SGX. Section 2.3 introduces an enclaved database with virtual memory support. The second part (chapter 3) is devoted to non-volatile memory technologies and persistent systems. Section 3.2 of this chapter presents a conception of hypervisor-based persistence and NV-Hypervisor. Chapter 4 concludes the thesis.

2. Memory encryption and trusted execution

Intel Software Guard eXtensions (SGX) and AMD Secure Encrypted Virtualisation (SEV) are new extensions of commodity server platforms which provide primitives for trusted execution in clouds. Both of these extensions offer system components which are inaccessible by privileged software and protected from attacks based on physical access. Physical memory pages of these components are encrypted, and processes of encryption and decryption are performed inside memory controllers without exposing encryption keys. As a result, a cloud user can use cloud infrastructures without risks of data breach.

While both platforms have many similarities, like memory encryption and isolation, they have different architectures. AMD SEV offers trusted computations at the level of virtual machines: a cloud user controls its own virtual machines, while a cloud provider controls the hypervisor and fundamental resources. The interaction between encrypted Virtual Machines (VMs) and the hypervisor is carried out similarly to ordinary Infrastructure-as-a-Service (IaaS) platforms. Programming of these encrypted VMs is also identical to that of ordinary VMs: a cloud user prepares a VM image and uploads it into the cloud.

Intel SGX, in turn, introduced a new system component named *enclave*. SGX enclaves only execute in the user-space and have unique features which differ the programming of enclaves from that of all other components of a system. Some of them are: trusted programs cannot issue system calls, and interaction between trusted and untrusted programs is performed in an Remote Procedure Call (RPC)-like way. These features, the programming models and system support of enclaves, are the primary topics of this chapter.

This chapter begins with the background section, which overviews modern platforms for cloud computing. In particular, this section compares two main platforms for trusted execution (Intel SGX and AMD SEV), describes challenges in the programming of enclaves, and provides an overview of related works which addressed these challenges earlier. Section 2.2 is devoted to the *E*Actors framework – an actor-based programming framework developed for SGX. Section 2.3 introduces STANlite – a database engine with virtual memory support developed for SGX-enabled cloud platforms.

2.1 Background

2.1.1 Modern platforms for trusted execution in clouds

Roots of trusted execution in clouds lie in areas of cryptography and developments of the Trusted Computing Group (TCG)[4]. Cryptography in the era of personal computers has enabled encryption of storing (DM-crypt[5], TrueCrypt[6]), transferring (SNP [45], SSL [46], TLS [47]) and in-memory (Armored [48], PRIME [49], Copker [50], Mimosa [51]) data. The TCG, in turn, standardised the conception of Trusted Computing and developed the architecture of a Trusted Platform Module (TPM)[7] – a tamper resistant security processor.

On top of this basis, Intel and AMD developed their own platforms for trusted execution in clouds. Both of these platforms use hardware-accelerated encrypted memory to protect user data, and complex infrastructures which remove software of a cloud provider from a Trusted Computing Base (TCB). However, despite the similarities, these platforms have many differences, which are considered in the following section.

2.1.1.1 AMD Secure Encrypted Virtualisation

AMD Secure Encrypted Virtualisation (SEV) is an extension which has been introduced as part of the Ryzen series of AMD CPUs [18] which aims to provide trusted execution in untrusted environments. This extension was released together with another extension – AMD Secure Memory Encryption (SME), which enables encryption of memory at page granularity. Because both of these technologies use hardware-accelerated memory encryption, it is reasonable to consider them together.

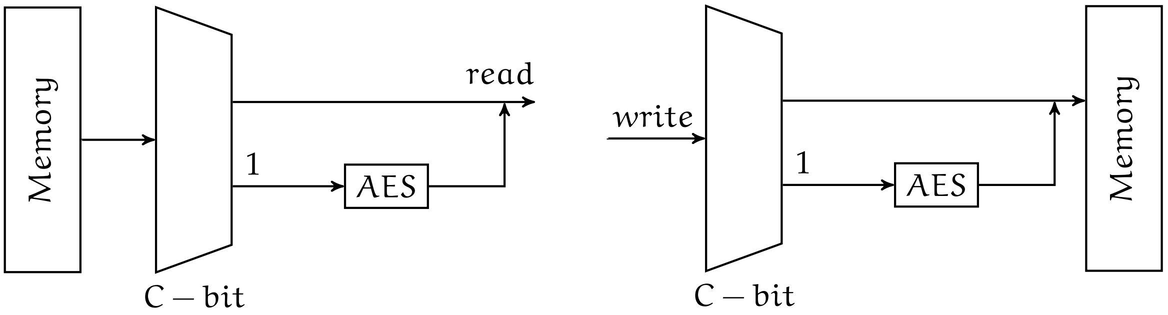
---

[4] Consortium developers of computing systems, https://trustedcomputinggroup.org

[5] http://www.saout.de/misc/dm-crypt

[6] http://truecrypt.sourceforge.net

[7] https://trustedcomputinggroup.org/resource/tpm-main-specification/



(a) Read access (b) Write access

Figure 2.1: Access to encrypted and non-encrypted memory pages in AMD SME

AMD Secure Encrypted Virtualisation (SEV) and Secure Memory Encryption (SME) have different purposes. SME is aimed at prevention of cold-boot attacks, while SEV is devoted to trusted execution in clouds. SME encrypts selected memory pages, and this process is controlled by the Operating System (OS) kernel. The kernel can specify which page needs to be encrypted by the modification of the `C` bit (number 47) in a Page Table Entry (PTE) which defines the corresponding mapping of physical and virtual pages. If the `C` bit is enabled for a page, then access to this page involves AES encryption/decryption (Figure 2.1). SME uses a single encryption key for all pages and Direct Memory Access (DMA) devices can access encrypted pages.

Secure Encrypted Virtualisation (SEV), in turn, disallows the hypervisor to control memory mappings. Each virtual machine is encrypted by its own encryption key, and the hypervisor can neither access it nor disable the encryption. Also, in contrast with SME, peripheral devices cannot access the memory of virtual machines. Interaction between the hypervisor and a VM is performed via a special type of encrypted memory, which is accessible by the hypervisor and the VM simultaneously. Migration of virtual machines is implemented via a built-in ARM-based security coprocessor, which establishes a connection between hosts and performs secure data migration without the involvement of the system software.

In sum, both technologies can be used for execution of encrypted VMs in clouds. However, these technologies have different threat models: SEV removes the hypervisor from the TCB, while SME relies on the privileged layer since it applies an encryption policy.

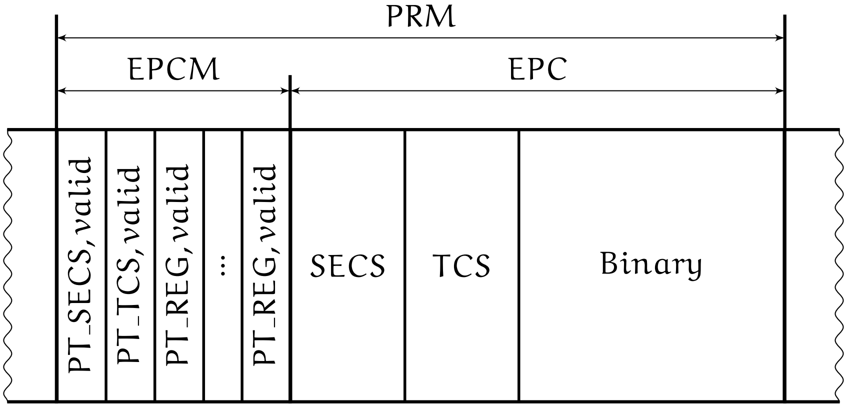


Figure 2.2: The structure of Processor Reserved Memory

2.1.1.2 Intel Software Guard eXtensions

The central component of the Intel SGX is an *enclave*. The enclave is an environment which includes data and code of a program. Creation and execution of enclaves is supported by new instructions of the Skylake microarchitecture.

All enclaves are located inside a physical memory region called Processor Reserved Memory (PRM). Neither system software nor peripheral DMA devices can directly access this memory region. Direct copying of data from memory modules also does not expose its content since the PRM data is stored in an encrypted form. Only a limited set of instructions is available for manipulation of the PRM data.

The PRM region hosts two elements. The first one, called Enclave Page Cache (EPC), consists of an enclave's management structures and execution entities (Figure 2.2). The second one, Enclave Page Cache Map (EPCM), is a structure, each element of which describes a single page of the EPC. Each element of the EPCM defines a type of the corresponding page inside the EPC.

There are several types of EPC pages [52]. Firstly, an EPC page can be used to store an enclave's metadata. Each enclave is described by a SGX Enclave Control Structure (SECS), which is stored inside a metadata page with the `SECS` type. A SECS page is used by SGX instructions only and cannot be mapped to virtual memory. A SECS defines a mode of operation (32 or 64 bits), and a set of processor features used by the corresponding enclave. An enclave can be created by calling the `ECREATE` instruction, and destroyed after removal of the corresponding SECS page from the EPC.

Secondly, a page can be used to store a Thread Control Structure (TCS). TCSs define the location of a *State Storage Area*, *Thread-local Storage* and some additional data. These structures are used to store contexts of threads executed inside an enclave. If a thread leaves an enclave, it saves its own context inside these structures and retrieves the context when it returns. Each executing thread inside an enclave consumes one TCS and thus, the number of TCS pages defines the number of threads which can be executed inside an enclave. TCS pages, as well as SECS pages, are inaccessible by software. However, pages which are used to store a context, like State Save Area (SSA) pages, are ordinary EPC pages and can be accessed by enclaved software.

The PRM region is limited in size. The maximum size of the PRM provided by market-available CPUs is 128 MiB. In addition, not all pages of the PRM are available to use by programs. The actual size of available memory is near 92 MiB [22].

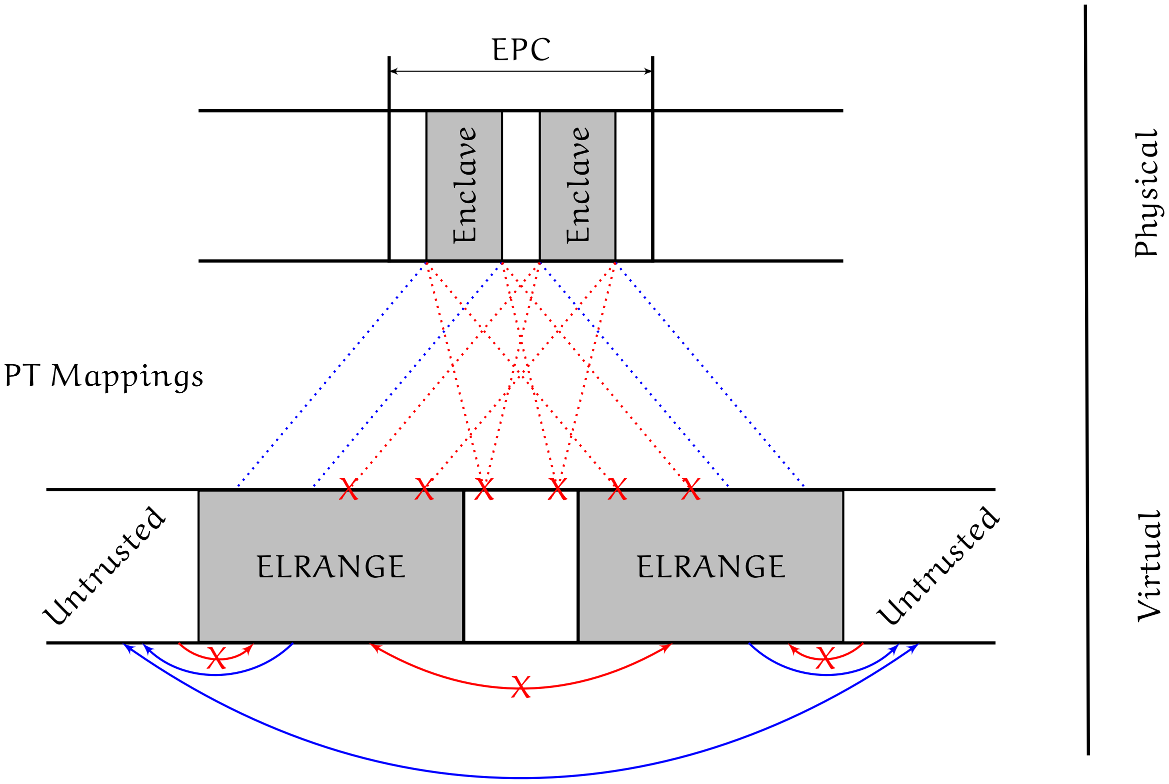


Figure 2.3: Mapping of EPC pages to Enclave Linear Address Ranges

Each enclave operates in a region of virtual memory named Enclave Linear Address Range (ELRANGE). Enclaved software can access the whole range of a process' virtual memory (except other enclaves), while non-enclaved software can access only virtual addresses beyond the ELRANGE borders (Figure 2.3). SGX maps pages stored inside an EPC into an ELRANGE. If the ELRANGE size is smaller or equal to the EPC size, then all physical pages are resident inside the EPC. However, the size of all ELRANGEs of all processes can be greater than the size of the EPC. In this case, the system software involves a paging mechanism, which extends the limits of the EPC by page swapping. EPC pages can be swapped out by calling of the `EWB` instruction and swapped in by calling of the `ELDU/ELDB` instructions. Swapped out pages are stored encrypted in memory outside the EPC.

Initial filling of an enclave by code/data pages is performed in the *initialisation* phase, after the creation of the enclave by the `ECREATE` instruction [53]. The `EADD` instruction performs creation of pages inside the enclave and fills them with the data taken from an untrusted virtual memory address range. For each new page, the system software invokes the `EEXTEND` instruction, which updates the enclave's *measurement* used in the software attestation process (described below). These operations can be performed only while the corresponding SECS is in the *uninitialised* state. This state changes to *initialised* after issuing of the `EINIT` instruction.

Attestation and data sealing

The Intel SGX architecture provides mechanisms which allow provisioning of secrets to an enclave. The first one is *attestation* – a mechanism for creating an authenticated assertion between two enclaves running on the same platform (*local attestation*). The architecture also offers an extended version of attestation, called *remote attestation*, which provides assertions to third parties located outside the platform. The second one is *sealing* – a mechanism for data encryption in a way which ensures that the data can be decrypted only inside another trusted enclave. This mechanism can be used to store secrets persistently when a host system is not active, or for the migration of secrets between different versions of enclaves.

Local attestation and data sealing rely on two measurement registers, MRENCLAVE and MRSIGNER. The first register provides the identity of an enclave. This identity includes the content of memory pages, order of the pages inside an enclave, and security properties associated with these pages. Technically, this value is a SHA-256 digest updated after each is added to the enclave page. The second register provides the sealer's identity. After the compilation of an enclave, a developer prepares a certificate (SIGSTRUCT), which includes the measurement of the enclave, a product identifier, and

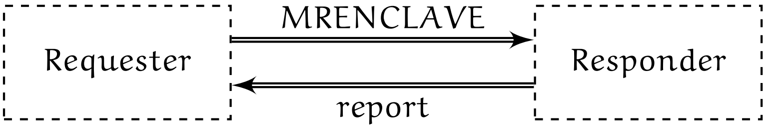


Figure 2.4: Local attestation of two enclaves

its own public RSA key. Then the developer signs this certificate and uploads it together with the enclave to the remote server[8]. During the booting of the enclave, the Intel SGX architecture verifies this certificate, compares the generated and the certificate's measurements, and fills MRSIGNER with a hash value of the public RSA key if the measurements are equal.

For data sealing, the Intel SGX architecture can base the encryption key on any of these registers. In the first case, the architecture generates sealed data which can be decrypted only inside the enclave that sealed it (or any other instance of the same enclave image). Other enclaves cannot do this since different enclaves have different MRENCLAVE values. In the second case, the architecture generates sealed data which can be decrypted only inside enclaves signed by the same RSA key.

Attestation demonstrates that an enclave is properly instantiated on the platform. In local attestation, this process involves two enclaves located on the same platform. Local attestation consists of two steps, which are depicted in Figure 2.4. During the first step, the requester enclave sends its own MRENCLAVE to the responder. Then, the responder prepares a report, generated by the `EREPORT` instruction together with the MRENCLAVE value. The report includes two identities of the enclave (MRENCLAVE and MRSIGNER), metadata of the enclave, user-provided data, and a Message Authentication Code (MAC) tag. The MAC is produced by a key called the "Report key", which is known only to the target (requester) enclave and to the `EREPORT` instruction. Then the responder sends the report to the requester. The requester retrieves its own "Report key", computes the MAC of the report and compares it with the MAC tag of the report. A match of these MACs confirms that both enclaves are run inside the same SGX-enabled platform. Then, the requester can compare MRENCLAVE of the responder enclave (retrieved from the report) with a pre-defined value to ensure that the responder enclave reflects the expected content.

---

[8] The Intel SGX Software Development Kit (SDK) embeds the certificate into the enclave's image

Remote attestation involves three parties [54]: a target server which hosts an enclave, an attestation service provided by Intel and a remote attester which wants to attest the enclave. Additionally, the target server needs to use two additional enclaves, provisioning and quoting. The provisioning enclave generates a certificate which proves that the target server is a proper SGX-enabled platform. This certificate is signed by the ”Provisioning Key” – a unique key fused and only known by Intel. When the attestation service verifies the certificate, it sends the ”Attestation Key”. This key is used later by the quoting enclave. The quoting enclave performs local attestation of the target enclave and prepares a *quote* – an attestation report signed by the ”Attestation Key” which can be verified outside the platform.

2.1.2 Challenges of SGX programming

From the software point of view, the use of enclaves is accompanied by several restrictions which impact the system performance and a programming model of enclaved programs. Below, these restrictions will be considered independently.

2.1.2.1 Self-contained environment and high transition costs

Both the code and data sections of an enclaved program are isolated from the untrusted environment. This means that the enclaved software cannot call code from shared libraries, or execute other programs. The enclaved program should be self-contained, otherwise it requires the use of heavyweight calls.

Communication between trusted and untrusted software is performed via special calls: *ECALL* and *OCALL*. ECALLs are used to enter into an enclave and can be called outside the enclave only. OCALLs, in turn, are used to exit an enclave and, as a consequence, can be called inside the enclave only. Any other mechanisms for communication, like traps or trampolines, are forbidden and cannot be used inside enclaves. These ECALLs and OCALLs are heavyweight and require at least 8000 CPU cycles [22]. Ordinary system calls, in contrast, require about 150 CPU cycles [21].

Thus, additional untrusted software should be involved for communication between an enclave and other components of a system including the kernel, or for calling of non-implemented inside enclave functions. For example, to send a message from an enclave to a pipe, an enclaved thread should firstly issue an OCALL and then, with the help of an untrusted library, it should perform a `write` operation on

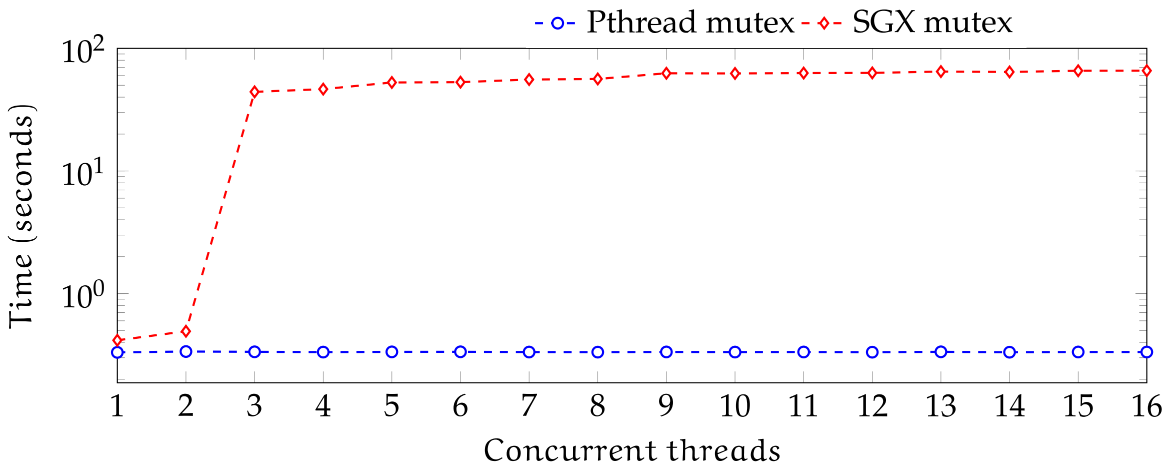


Figure 2.5: Synchronisation costs of pthread mutex versus SGX SDK mutex.

the pipe's descriptor. After that, the thread can return into the enclave, and for that it issues a heavyweight ECALL. Additionally, if the target component is not a part of untrusted software and located inside another enclave, the number of issued ECALLs and OCALLs doubles.

Synchronisation primitives like pthread's mutexes also cannot be used inside enclaves. An enclaved thread cannot request the kernel to suspend it inside an enclave. Instead, the thread has to issue an ECALL, or use spin-lock based synchronisation. In fact, the synchronisation solution offered by the SGX SDK combines both approaches. If two enclaved threads try to access shared data concurrently, they both need to:

- perform a spin-lock based attempt to acquire a lock
- leave the enclave if several iterations of accruing were unsuccessful
- lock a pthread mutex (or become suspended and then lock the mutex)
- enter back to modify shared data
- leave the enclave to unlock the mutex
- enter back to continue the own execution

Figure 2.5 shows the difference between two synchronisation mechanisms: an ordinary pthread mutex, and an SGX mutex – the synchronisation primitive provided by the Intel SGX SDK. The figure shows the time necessary to dequeue 10^6 elements by the different number of concurrent threads. As can be seen, synchronisations are very costly and up to 200× slower compared to non-enclaved synchronisation primitives.

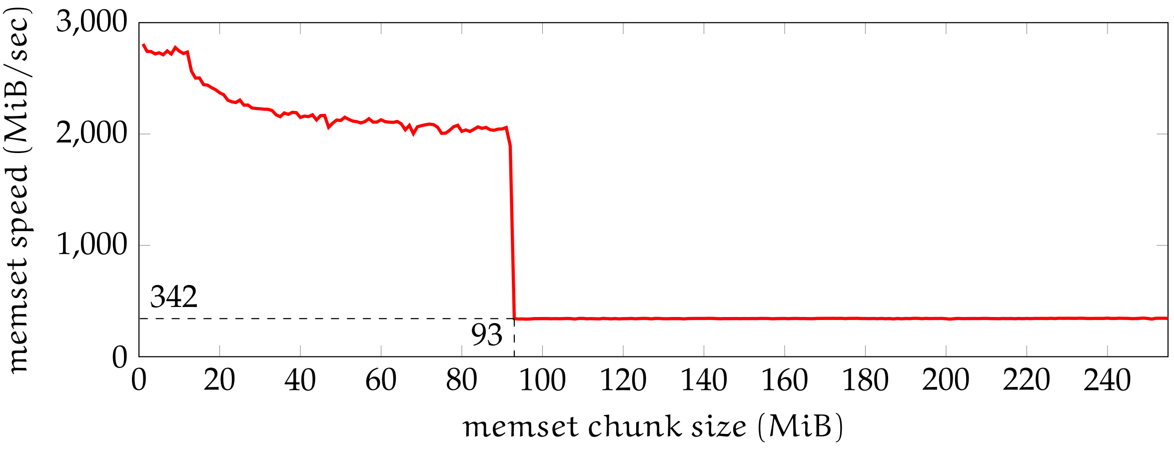


Figure 2.6: Performance of in-enclave memset operation

2.1.2.2 EPC paging

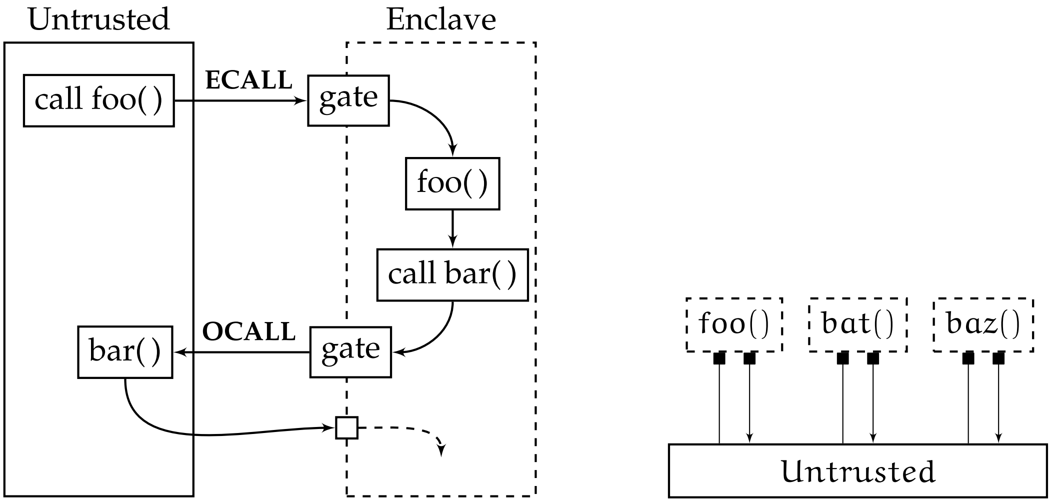
Page eviction takes place when the size of memory used by all enclaves exceeds the EPC size. Within this process, the SGX driver encrypts some pages located inside the EPC and evicts them into memory located outside of the PRM. The SGX paging is heavyweight similar to ECALLs, and has a dramatic impact the performance of enclaved software.

Figure 2.6 shows the impact of paging on the performance of a microbenchmark. The simple *memset*[9] function is used inside a single enclave with one TCS to fill various sizes of memory, started from a chunk 1 MiB in size and up to a chunk 256 MiB in size. The number of bytes cleared per second is used as the performance metric. As can be seen, after reaching the EPC border (≈92 MiB), the number of bytes filled per second decreases 6 times approximately.

There are several reasons for this performance degradation. Firstly, eviction of a page causes flushing of all cached address translations on all logical processors [55]. This also causes Asynchronous Enclave Exit (AEX) on all threads which share the same enclave on all CPU cores. Secondly, since eviction of a page is performed to untrusted memory, the SGX engine encrypts swapped out pages. Finally, to prevent rollback and replay attacks, the SGX engine saves *nonces* in *Version Arrays* – regions of EPC memory which store versions of evicted pages. On the swap-in operation, the SGX engine ensures that only the last evicted version of a page can be loaded.

---

[9] In accordance with optimisation flags, a compiler uses different implementations of basic functions like *memset* or *memcpy*. The performance of these functions plays an important role in low-level measurements. In all experiments and evaluations, the `-O2` compiler optimisation flag was used.



(a) Invocation of enclaved `foo()` and non-enclaved `bar()` (b) Communication with different enclaves

Figure 2.7: E(O)CALLs-based invocation of functions

2.1.3 Related works

While Intel SGX is relatively new technology, it was considered in various research works. Relevant to the research question of this thesis, the works can be divided into two groups. The first group is devoted to programming approaches of enclaves. The second covers system support of enclaves, i.e. aspects of security and performance. Below, these works will be considered independently.

2.1.3.1 Programming approaches

The spectrum of the enclave's programming approaches is located between two extreme cases. The first one is represented by the Intel SGX SDK [56], which enables programming of enclaves in an RPC-like manner. In this approach, an enclave works as a TPM. However, in contrast with a hardware TPM, this "software TPM" can be uploaded, and a single hardware platform can run multiple "TPMs". From the software point of view, programming approaches of enclaves and TPMs are quite similar: an enclave includes several trusted functions and these can be called by untrusted software (Figure 2.7). Transitions between trusted and untrusted software are performed via heavyweight ECALLs and the SDK provides an Enclave Description Language (EDL), which helps a developer to design and implement an RPC-like interface. In this approach, enclaved software is characterised by an extremely small TCB and provides primitive functions developed especially for enclaves.

The second case is represented by various projects based on the idea that enclaves are environments for the trusted execution of complex services. These services should not be developed from scratch for enclaves but rather be ported to enclaves. However, since the execution of programs inside enclaves differs from the execution of ordinary processes, the enclaved software requires additional support provided by frameworks. A framework should implement missing mechanisms, such as the interaction of enclaved software with devices or system components. Frameworks and projects like Haven [26], SCONE [25], Graphene-SGX [27] and Panoply [57] follow this programming approach, but have different designs and implementations.

As mentioned previously, an enclaved program cannot use system calls but can use heavyweight alternatives called OCALLs. All non-supported functions can be mapped to OCALLs, but it is inefficient because of the low performance of OCALLs. If enclaved software emulates some frequently used system calls, then the number of issued heavyweight OCALLs decreases. However, this approach increases the TCB size of enclaved software and thus, there is a tradeoff between the TCB size of enclaved software and its performance. One can compare the frameworks from the point of view of this tradeoff.

Haven [26] offered to use a full-fledged libOS inside enclaves. This libOS implemented a maximum possible number of system calls and issued OCALLs only when an operation could not be emulated inside an enclave. Haven shifted the tradeoff to the side with low-intensive calls and a huge TCB. Graphene-SGX [27] followed the same design but used the glibc library. SCONE [25] tried to minimise a TCB without increasing the OCALL intensity. For that, SCONE used the lightweight *musl* library and an asynchronous interface, which was based on queues [58] for communication of trusted software and untrusted helpers. To prevent attacks of system software on enclaved programs[10], both SCONE and Haven checked the correctness of the values returned by the untrusted helpers. Panoply [57] used synchronous calls, removed the `libc` library from a TCB and mapped the POSIX interface to OCALLs. Thus, Panoply shifted the tradeoff to the side with high-intensive calls and a minimalistic TCB.

---

[10] So called Iago attack [59]

2.1.3.2 System support

Different works considered weaknesses of enclaves and developed system support for them. These works can be separated into several groups.

Attacks

Various works considered security aspects of enclaves. Xu et al. [60] introduced a SGX-specific type of side-channel attack called *a controlled-channel* attack. Enclaves cannot control access rights of their own pages, and the malicious kernel can manipulate these rights to enable page faults inside enclaves. By this, firstly, the kernel can track sequences of page faults and thereby reconstruct the execution control flow, which is possible even at instruction-level granularity [61]. Secondly, an attacker can also retrieve enclaved data, and even extract cryptographic key bits from unmodified versions of OpenSSL and Libgcrypt [61]. Thirdly, this approach can be used to stop the execution of particular threads inside an enclave and thereby exploit *use-after-free* and *time-of-check-to-time-of-use* bugs [62] in enclaved software. Enclaves can be protected from these controlled-channel attacks by a software-only defence technique, but with high performance overhead [63].

Additionally, enclaves are vulnerable to ordinary side-channel attacks[11]. Lee et al. [64] implemented a *branch shadowing* attack which identified execution flows of enclaved software. Schwarz et al. [65] and Moghimi et al. [66] demonstrated successful Prime+Probe cache attacks on vulnerable cryptography libraries. These attacks required an interrupt of execution of victim processes, which can be detected [67, 68]. However, Brasser et al. [69] demonstrated an attack which does not require the interruptions.

Protection

Another group of works were aimed at the protection of enclaved software from external attacks and exploitation of internal bugs. Chandra at al. introduced [70] a defence strategy with reasonable performance degradation which added random memory access to the existing program's execution flow. These accesses added noises to the execution pattern and made retrieving of the execution flow complicated. ZeroTrace [71] presented more sophisticated oblivious memory services based on an

---

[11] Even to Spectre-like: https://github.com/lsds/spectre-attack-sgx

oblivious block-level memory controller. SGXBounds [72] introduced an LLVM-based compiler framework which added memory-safety for enclaved software written in unsafe languages like C and C++. The framework uses the unused upper 32 bits of 64-bit pointers to store tags which define bounds of referenced objects and location of additional metadata. SGXBounds uses these tags for runtime bounds checks of memory access operations like store, load, etc. SGX-Shield [73] implemented randomisation of in-enclave address space.

Performance

The research community also addressed the performance issues of enclaved software. HotCalls [21] offered a hand-crafted *busy-wait* mechanism which enabled concurrent access to shared objects inside an enclave without the use of sgx_mutexes. Switchless Calls [74, 75] introduced a similar *busy-wait* technique for asynchronous invocations of enclaved functions. Eleos [22] introduced a programming language-based paging mechanism which aimed at avoiding costly enclave exits. Vault [76] also addressed the problem of memory paging and proposed *Variable Arity Unified encrypted-Leaf Tree* – a data organisation structure which reduced the integrity tree storage overhead and significantly decreased the impact of the EPC paging.

Functionality

Several works introduced a new functionality of enclaves. The hypervisor and the kernel cannot access pages of enclaves and thus, they cannot provide enclave migration. To overcome this drawback, Gu et al. developed [77] a software-based protocol which implemented migration by enclaved software. Glamdring [78] introduced an automatic source-level partitioning framework that enabled a split of applications written in C into multiple parts, and secure them by enclaves. Weichbrodt et al. presented [79] a framework for performance analysis of enclaved software.

2.1.4 Conclusion

Two mainstream platforms for cloud computing have extensions for trusted execution. AMD SEV offers trusted execution in the form of encrypted virtual machines, while Intel offers the same in the form of new system components called SGX enclaves. Enclaves are parts of user-space programs which are inaccessible by the privileged software, other enclaves, and devices.

The programming of enclaves is accompanied by several challenges: enclaved programs cannot have dependencies outside their own enclaves, transitions between enclaves and untrusted software are very costly, and the performance of enclaved programs can dramatically degrade because of page swapping. These challenges were addressed by various research works aimed at the development of new programming models and system support for enclaves. For example, the first and second challenges were addressed in frameworks such as Haven [26], SCONE [25], Graphene-SGX [27] and others. The frameworks enable execution of legacy programs inside enclaves and offer primitives to reduce the transition costs. However, these projects do not fully use the advantages of enclaves since all of them offer monolithic single enclave solutions. At the same time, the Intel SGX architecture enables execution of multiple enclaves inside a single process, thereby allowing development of applications which have multiple trusted components. This programming model and the corresponding use cases are considered in the following section.

The third challenge was addressed in the Eleos [22] framework. This framework can mitigate the impact of hardware-based paging by the use of in-enclave software-based paging. However, legacy programs cannot profit from this framework, since it requires the redevelopment of the whole software. For some software systems, like databases, this approach cannot be applied because of high software complexity or incompatibility of the programming languages. Such programs require another approach, which is considered in section 2.3.

2.2 *E*Actors: an actor-based framework for trusted execution

Section 2.1 provided an overview of the Intel SGX architecture and various related research projects. As shown, SGX enclaves are parts of user-space programs which are inaccessible by privileged software, other enclaves or devices: only code located inside the enclave can process data located inside the same enclave. Enclaves have several shortcomings: transitions between enclaves and untrusted hardware are high and memory used by all enclaves without page swapping is quite small. Despite this, SGX enclaves enable trusted execution in untrusted environments and have several additional features. One of them is the possibility to partition a single program into multiple enclaves.

Indeed, a single process can host multiple enclaves, and therefore can isolate and protect parts of the application from each other. In the case of intrusion into an enclave, only code and data located inside this enclave will be compromised, while other enclaves will be protected. An example of the applicability of such design can be a secure instant messaging service with protected group chats. The processing of messages inside different enclaves protects them, even if one enclave, or an untrusted part of the application, is compromised.

The programming environment which is used to implement such use cases should meet several requirements. Firstly, partitions of the application should have the minimal TCB and different code bases. The huge TCB increases the probability of defects [80], while the code reuse makes all enclaves vulnerable if the code has a flaw. Secondly, the environment should provide an efficient mechanism for data exchange between partitions. In addition to this, the environment should be aware of the aforementioned shortcomings of enclaves.

Based on section 2.1, one can identify two generalised programming approaches for development of enclaved applications. The first one is offered by the Intel SGX SDK. This SDK provides primitives for co-development of trusted and untrusted programs: a developer defines functions which should be located inside enclaves, and then can invoke them in an RPC-like way. While the SDK provides the necessary instruments to program enclaves, this programming approach has several shortcomings from the point of view of multi-enclave applications:

Costly enclave interaction: The SGX SDK offers an Enclave Description Language (EDL) to define interfaces between enclaves and untrusted software. These interfaces use heavyweight ECALLs and OCALLs, which are more than 50 times slower than ordinary system calls [74].

Hardcoded partitioning: The code generation process of the SGX SDK requires explicit definition of which code needs to be located inside a trusted area, and which needs to be located inside an untrusted one. This separation limits a developer in further design and deployment decisions.

Costly synchronisations: The SGX SDK offers a mutex-like synchronisation primitive which allows synchronising access to shared data (section 2.1.2.1). However, threads cannot be suspended by the kernel inside an enclave when waiting for a condition to be fulfilled. As a consequence, either spin-locking needs to be performed or the thread has to leave the enclave, which requires the use of heavyweight ECALLs and OCALLs.

The second programming approach is proposed by projects like Haven [26], SCONE [25], Graphene-SGX [27] and Panoply [57]. In these projects, an enclave is considered as an environment for execution of legacy programs. A tiny shim layer, or a library OS, is located inside an enclave and provides a kernel (or more high level) Application Programming Interface (API) to an enclaved program. Only a small number of functions can be fully implemented inside an enclave, and thus, this layer should interact with untrusted helpers. In the context of multi-enclave programming, this approach increases the TCB size of enclaved partitions, only partly[12] solves the issues of costly synchronisations and transitions, does not have fast inter-enclave communication primitives, and has a hardcoded monolithic partitioning.

This section presents an alternative programming approach based on ideas of *actors*. An actor as a computation entity which can, in reaction to a received message, send messages to other actors [81]. As a result, multiple actors can work concurrently and do not possess a shared execution state, which avoids the use of heavy synchronisation primitives (section 2.1.2) and prevents exploitation of synchronisation-based bugs [62]. This programming model requires the development of a special programming framework, since existing ones, such as the CAF framework [82] or Akka [83], do

---

[12] SCONE offers an asynchronous interface and user-level threading

not support SGX enclaves. These frameworks are not designed for trusted execution and have heavyweight runtimes, the porting of which to enclaves is challenging.

This section introduces the *E*Actors framework – a programming framework which inherits the actor-based programming model and is built from scratch in accordance with Intel SGX features. The section is organised as follows. Section 2.2.1 provides the general overview of the framework: the programming model, definition of entities, life-cycles of actors and more. Section 2.2.2 covers architectural and implementation details of various components, such as communication primitives and system actors. Section 2.2.3 is devoted to architecture of use cases: a microbenchmark, a secure multi-party computation service, and an *E*Actors-based XMPP messaging service. Section 2.2.4 presents evaluation of the framework and the use cases. Section 2.2.5 discusses related works, while section 2.2.6 summarises the section.

2.2.1 General architecture and basic primitives

The *E*Actors framework offers an actor-inspired programming model. An actor is a computation entity which can, in reaction to a received message, send messages to other actors. The framework has the following design goals in response to the identified shortcomings of the SGX SDK and existing frameworks:

> **Messaging** The *E*Actors framework enables fast message exchange between $_E$actors[13]. This especially applies for $_E$actors communication across enclave boundaries, which facilitates the use of multiple enclaves.
>
> **Lock avoidance** The *E*Actors programming model avoids costly synchronisation because actors do not rely on a shared state but instead exchange messages.
>
> **Flexibility** The *E*Actors framework offers a flexible use of trusted execution. It separates the code of an $_E$actor, which is implemented in a standard programming language, from the associated deployment policy. This policy defines the assignment of a given $_E$actor to computational resources (i.e., CPU cores, threads) and especially to enclaves. This is facilitated by providing uniform communication primitives which transparently select adequate communication mechanisms, regardless where an $_E$actor is placed (i.e., inside or outside of an enclave) or

---

[13] The notion of actors inside the *E*Actors framework

```
struct state {struct channel chan[2];int first;}

void aping(struct actor* self) {
  if(self->state->first) {
    self->state->first = 0;
  } else {
      /* receive a pong */
    char* msg = recv(&self->channel[0]);
    if(msg == NULL)
      return;
  }
  /* send a ping */
  send(&self->channel[1], "ping");
}

void aping_ctr(struct actor* self) {
  self->state->first = 1;
  connect(self->channel[0]);
}
```

Listing (2.1) Pseudo-Code of an ${}_{e}$`actor`

with whom it communicates. As a result, an ${}_{e}$`actor` can be deployed either in or outside of an enclave without further modifications to its application logic.

The reminder of this subsection provides an overview of each of these goals and demonstrates how they are applied in the framework.

2.2.1.1 *E*Actors programming model

The *E*Actors framework features a lean actor programming model. In order to implement an ${}_{e}$`actor`, a developer has to provide the body function of the actor, which contains the application logic and a constructor function. The purpose of the latter is to initialise communication channels to other ${}_{e}$`actors` and initialise the private state of the ${}_{e}$`actor` at startup time. Thus, connections are in essence statically assigned to avoid additional naming and resolving mechanisms with the aim of a small TCB.

Listing 2.1 presents a simplified example of an ${}_{e}$`actor` written in C. The `PING` ${}_{e}$`actor` sends a `ping` message when it receives a `pong` message from a `PONG` ${}_{e}$`actor`. The latter works analogously to the presented example. The structure `state` declares the private state of the actor, the `aping` function is the *body* function, and the `aping_ctr` is the *constructor*. Once called by the runtime, the constructor function initialises

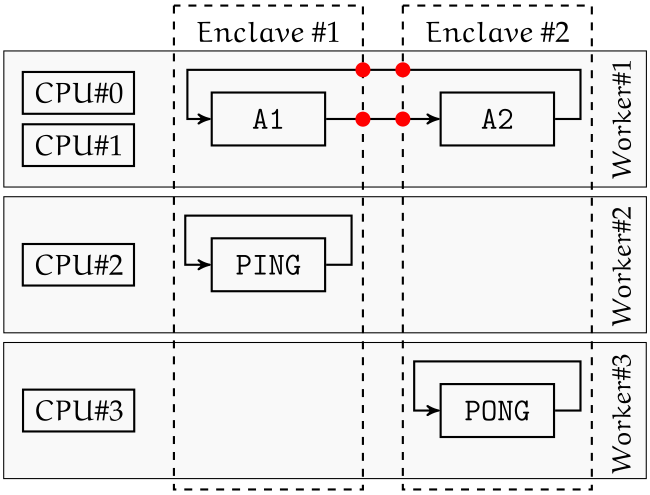


Figure 2.9: Deployment of $_e$actors, workers, and enclaves

the field `first` to 1 (line 17), and connects the `PING` $_e$actor to a `PONG` $_e$actor via the communication channel 0 provided by the runtime (line 18). Next, the *E*Actors runtime regularly executes the body of the $_e$actor. When the runtime executes `aping` for the first time (line 4), `aping` simply generates an initial `ping` message (line 13) in order to start the `ping`/`pong` message exchange. Each time the body function `aping` of the $_e$actor is executed, either a message is received via the channel and `ping` is emitted, or the $_e$actor simply *returns* because no data needs to be processed.

2.2.1.2 *E*Actors runtime

At its core, the *E*Actors runtime enables mapping of computational resources in terms of CPUs and threads to $_e$actors. More importantly, it allows defining whether an $_e$actor should be executed in a trusted execution context (i.e. an enclave) or as a part of the untrusted application. As a result, the use of multiple enclaves is supported.

Figure 2.9 illustrates an example deployment. It defines $_e$actors (`A1`, `A2`, `PING` and `PONG`) and two enclaves. The `A1` and the `PING` $_e$actors are located in the first enclave, while the `A2` and the `PONG` $_e$actors are located in the second enclave. To execute these $_e$actors, three *workers* are utilised.

A worker is a framework abstraction to manage a POSIX thread. The first worker is bound to CPUs 0 and 1, and executes the `A1` and the `A2` $_e$actors in round-robin. The second worker is bound to CPU 2 and executes only the `PING` $_e$actor, while the third worker is bound to CPU 4 and executes only the `PONG` $_e$actor. If all $_e$actors assigned to a worker are confined to the same enclave, the worker does not leave the enclave (e.g. `Worker#2` for the `PING` ${}_{e}$actor). Otherwise, as in case of the `A1` and the `A2` ${}_{e}$actors, the worker has to migrate from enclave to enclave in order to execute the body functions, which will result in costly execution mode transitions. Such an approach should usually be avoided but can be used if ${}_{e}$actors are dispersed over multiple enclaves and infrequently activated.

Infrequently activated actors can use another feature of the framework: temporary self-deactivation. Enclaved ${}_{e}$actors cannot be suspended inside enclaves and thus, the CPU becomes overloaded by such ${}_{e}$actors, which unsuccessfully try to dequeue incoming messages in most cases. To prevent overloading, an ${}_{e}$actor can request the runtime to disable its own invocation for some invocation cycles. During this time, the corresponding worker invokes other ${}_{e}$actors, or leaves the enclave for a pre-defined short period if all its own ${}_{e}$actors are self-deactivated. This technique significantly decreases the performance overhead caused by idle ${}_{e}$actors and contributes to scalability of framework-based applications.

To implement the outlined scenario, plus more complex ones, a developer defines the necessary mapping of computational resources and trusted execution contexts of ${}_{e}$actors in a special configuration file. This file builds the basis for a custom build process and leads to the generation of the source tree of a project. The tree includes source code of all actors and components of the framework. Compiled together with the SGX SDK, the source tree generates binaries with untrusted applications and enclaves that implement the envisioned deployment. During this process, the building system also extracts *measurements*[14] from the compiled enclaves and embeds these values into structures responsible for local attestation and establishment of encrypted connections. Section 2.2.2.1 and section 2.2.3.2 describe this process in detail.

When the application is started, the generated *E*Actors runtime creates the enclaves, allocates private memory regions, calls the constructors of the actors, and creates, as well as starts, the workers. Each worker then executes the body functions of its assigned ${}_{e}$actors.

---

[14] See section 2.1.1.2

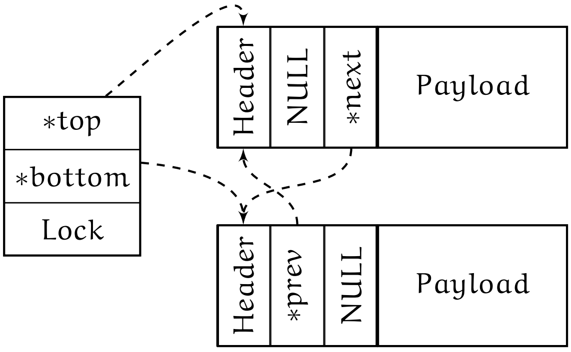


Figure 2.10: Double-linked list of nodes

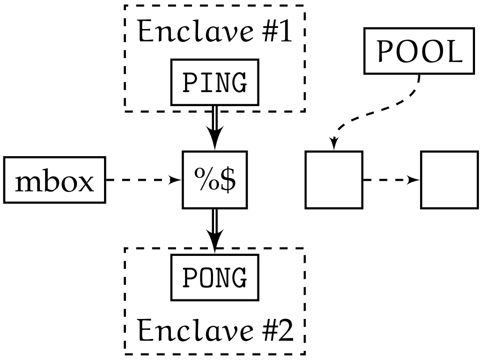


Figure 2.11: Message exchange between two ${}_{e}$actors

2.2.1.3 Memory management and messaging

As mentioned previously, ${}_{e}$`actors` use messages for communication. To provide this, the *E*Actors framework has three basic primitives called `pools`, `mboxes` and `nodes`. A `node` is a memory object which consists of two elements: a *header* and a *payload*. The payload is a memory region used to transfer ${}_{e}$`actors` messages. The header consists of multiple data pointers to manage nodes. A `pool` is an abstraction which refers to a set of empty `nodes`. The framework preallocates private and public `pools` at system start. A `mbox` is an abstraction which refers to a set of linked nodes used for message exchange. `Mboxes` and `pools` are organised in the form of bi-direction double linked lists implemented on top of Hardware Lock Elision (HLE) [84] (Figure 2.10). They have different APIs and semantics: `mboxes` offer FIFO semantic, while `pools` implement LIFO semantic.

To send a message, an ${}_{e}$`actor` needs to dequeue an empty `node` from a `pool`, fill its payload, and enqueue it to a `FIFO` `mbox` (Figure 2.11). At the same time, another ${}_{e}$`actor`, tries to dequeue the message `node` from the `mbox`. Upon success, the second ${}_{e}$`actor` retrieves the payload and can return the used `node` back to the `pool`.

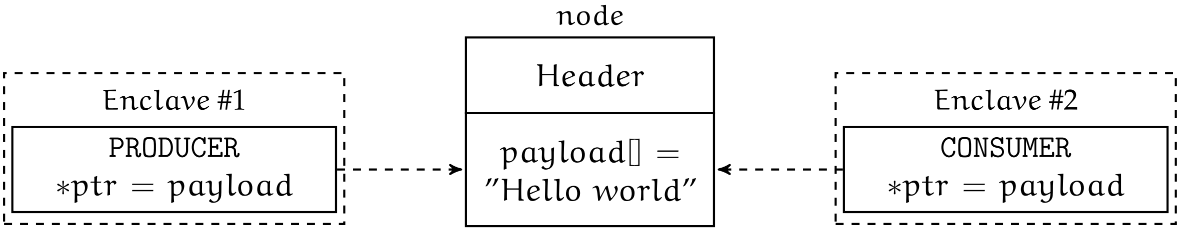
2.2.2 Design and implementation of system components

To facilitate applicability, scalability and flexibility of ${}_{e}$`actors`-based applications, the *E*Actors framework provides the support of three high-level components: a unified communication layer, system actors, and an *E*actors Object Store (EOS). The following subsections consider these components.

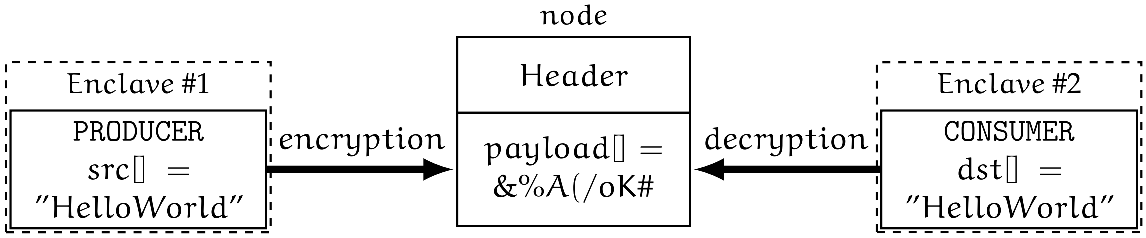
2.2.2.1 Connectors and cargos

`Mbox` is a universal primitive for communication of $_e$actors. These $_e$actors can be located inside a single or multiple enclaves and thus, communication can be *public* or *private*. Public communication uses `pools` and `mboxes` located inside untrusted memory, while private communication uses the same located inside an enclave. Additionally, communication can be encrypted and non-encrypted.

Assigning of actors to enclaves is defined by a configuration file and thus, switching between encrypted/non-encrypted and public/private implementation should be hidden from the actor's source code. `Mboxes` cannot provide the necessary unified communication layer because encrypted and plaintext data are transferred in different ways.



(a) Plaintext data transfer



(b) Encrypted data transfer

Figure 2.12: Different forms of data transfer

As mentioned previously, to transfer a plaintext message from one $_e$actor (producer) to another $_e$actor (consumer), the producer needs to pop an empty message `node` from a `pool`, fill the payload of the `node`, and then push the `node` into a `mbox` of the consumer. The consumer, in turn, needs to pop the message from the `mbox`, and then it can access the payload. Data access of both participants is performed via pointers: both the producer and the consumer can access the payload of the `node` directly (Figure 2.12a). In sum, the single data transfer requires one memcpy operation (to fill the payload), and four enqueue/dequeue operations (to allocate, to send, to receive and to return).

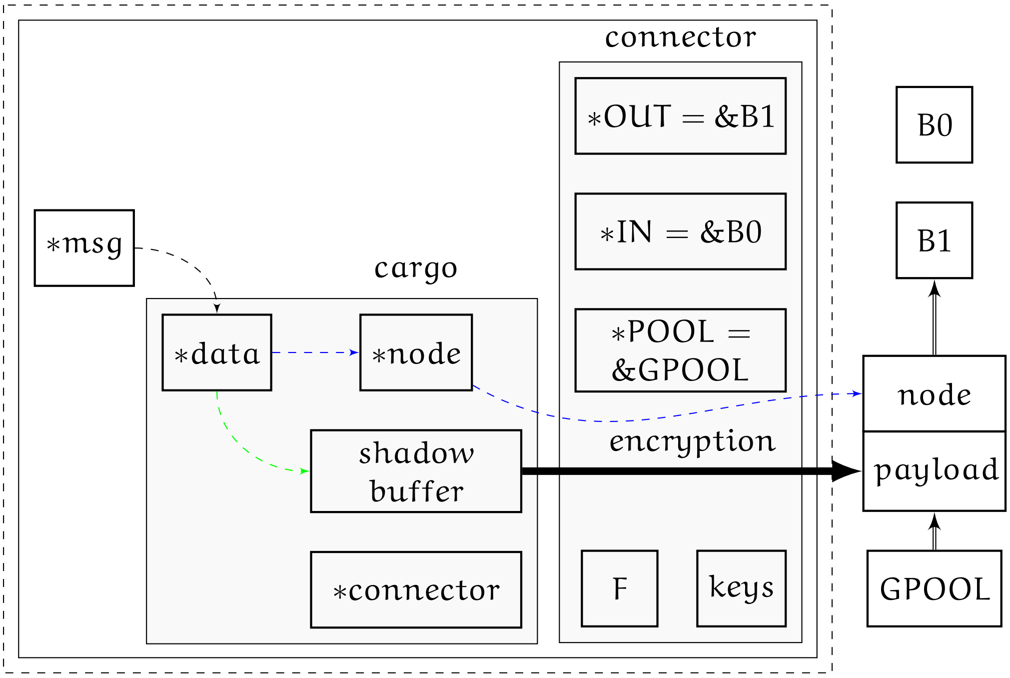


Figure 2.13: Internal structure of a connector and a cargo

The data transfer of an encrypted message is different. A fresh node obtained by the pop operation cannot be used directly as a non-encrypted one can. Instead, the producer needs to allocate a shadow buffer (src[ ]) in the trusted address space, place data to send ("Hello world") within, and then encrypt the data into the node's payload (Figure 2.12b). Upon receiving, the consumer also needs to allocate a shadow buffer (dst[ ]) inside its own trusted address space and within it, decrypt the payload of the incoming message.

This need for the shadow buffer allocation makes use of `mbox` inflexible because different types of communication require different methods and data structures. To overcome this inflexibility, the *connectors* and *cargos* were developed.

Design

The *connector* is an abstraction which describes a unified communication layer between two actors. The *cargo* is an object which carries a message from one actor to another via a connector. Connectors can *create*, *send*, *receive* and *return* cargos. These entities are considered in detail below.

A connector consists of several components (Figure 2.13). Firstly, it includes two pointers to `mboxes` (`*IN` and `*OUT`), and one pointer to a `pool` (`*GPOOL`). These `mboxes` and the `pool` should be accessible by both actors involved in message exchange. Secondly, a connector includes an encryption context (`key`). This contains encryption keys used for message encryption. A connector also includes a flag field (`F`), which defines the type of the connector: plaintext or encrypted.

The cargo is a compound object which contains a message. Firstly, a cargo includes the pointer to the *connector* which produced it (`*connector`). A cargo can be sent or returned only by the connector which created the cargo. Secondly, a cargo has a pointer to a `node` used for a data transfer (`*node`). Thirdly, a cargo has a data pointer `*data`, which points to the node's payload or to the *shadow buffer*.

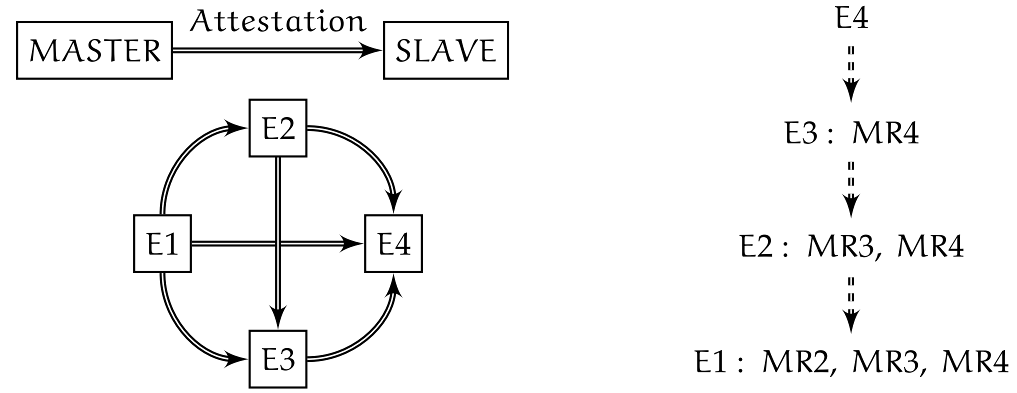
If a connector has an encryption flag, it produces encrypted cargos. In an encrypted cargo, the `*data` points to the shadow buffer (green line), which is allocated inside trusted memory. The outgoing data will be taken from this shadow buffer and encrypted into a message node when the cargo is sent. In contrast, when the connector receives a cargo, it decrypts the content into its own shadow buffer. Plaintext connectors, in turn, produce plaintext cargos, and data pointers of these directly point to a payload of their own nodes (blue line).

Types of connectors, as well as pointers to pools and mboxes, are provided by the developer at compilation time. These configurations are assigned to connectors at the construction phase and should not be changed during the execution. Manipulations with internal shadow buffers and pointers are transparent for a developer and $_{e}$actors. As a consequence, this approach does not require modification of the $_{e}$actor's body function when the type of a connector changes. A developer only needs to change the type of a connector, which can be done in the $_{e}$actor's constructor.

Key-exchange procedure

The framework uses the AES-128-GCM algorithm with an incremental initialisation vector to message encryption. Thus, each pair of encrypted connectors needs to have the same pair of 128-bit variables, an encryption key and an initialisation vector. These are generated during the special key-exchange procedure based on *local attestation*.

The key-exchange procedure involves two connectors and is performed by constructors of actors located inside different enclaves. The first constructor and its connector have the MASTER role, while the second constructor and its connector have the SLAVE role. Constructors exchange messages via these connectors as follows:



(a) Possible combinations of local attestation (b) Compilation order

Figure 2.14: Chaining of local attestation

1. MASTER generates a pair of keys used for AES-GCM, retrieves the *measurement* of its own enclave and sends it to SLAVE.
2. SLAVE generates a pair of RSA keys, receives the measurement and generates a local attestation report with this measurement. This report also includes a hash of the public RSA key. Then SLAVE sends the report with the public key to MASTER.
3. MASTER receives, verifies the report, and compares the SLAVE's measurement with the expected value. If the these measurements match, then MASTER encrypts by the public RSA key the AES-GCM keys, and sends them to SLAVE.
4. SLAVE receives the encrypted message, decrypts it and initialises the encryption context of the connector.

The building system automatically generates measurements of all (except one) enclaves. However, since these values can be extracted only from compiled enclaves, the building system cannot provide the same table with measurements for all enclaves: the first compiled enclave cannot have the measurement of the second compiled enclave if the second compiled enclave includes the measurement of the first enclave. In other words, in a pair of enclaves, only one enclave can attest the other one. Meanwhile, the building process enables chaining of attestation, as depicted in Figure 2.14. The order of the chain is defined by the order in which enclaves are compiled. For example, the framework firstly compiles the E4 enclave, and thus, stored inside it connectors can have only the `SLAVE` role. The last compiled enclave (E1) includes measurements of all other enclaves (MR2, MR3, MR4), therefore, its actors can attest all other enclaves.

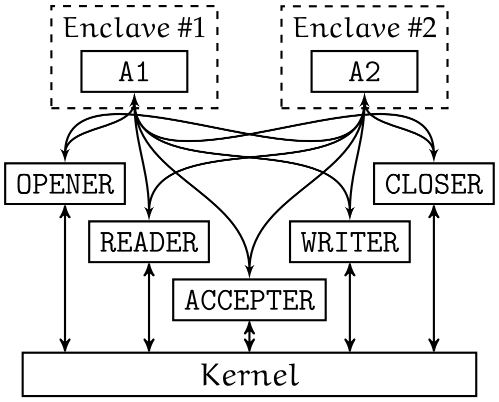


Figure 2.15: Interactions of enclaved actors with network system actors

2.2.2.2 System actors

Enclaved $_e$`actors` cannot directly interact with the kernel. This requires the use of ECALLs and OCALLs, which are forbidden in the framework. Instead, this functionality is provided by *system* actors.

System actors are the special actors which execute in the untrusted area and provide *mechanisms* – system calls. The framework provides several system actors, which can be grouped by subsystems: networking (`socket()`, `bind()`), memory (`mmap()`, `munmap()`, `mprotect()`), input/output (`open()`, `close()`, `read()`, `write()`, etc.). Each system actor provides only one mechanism and follows a similar pattern of work: a system actor receives a specially prepared message from $_e$`actors`, performs its mechanism on arguments stored inside the message, and sends the results back. The message includes arguments for the mechanism and a `mbox` where results should be returned. To decrease the overhead caused by the intensive messaging, system actors support batched requests.

Two features of system actors increase their performance and scalability. Firstly, all system actors are stateless actors, i.e. they do not store any data between invocations. Thus, any system actor can handle any request from any other $_e$`actor`. As a consequence, the framework can be configured to spawn multiple system actors, attached to different workers, with the same mechanisms. Secondly, all system actors use non-blocking calls. In particular, for network actors, this means, that instead of waiting for an incoming message via the `select()` or the `poll()` system calls (and thus block the worker), a network actor can access sockets by non-blocking `read()`, and if there is no incoming event, the actor indicates this in the return message.

Scalability of network actors

The design of system actors enables communication of trusted actors with clients over connection-agnostic network actors (Figure 2.15). In accordance with performance requirements, a developer can spawn any number of network actors assigned to any number of workers. For example, if a system needs to consume more network traffic than what is produced, the framework can be configured to spawn more `READER` actors than `WRITER` actors. This approach is used in the XMPP service (section 2.2.3.3) to increase the performance of a network-intensive application.

2.2.2.3 *E*actors Object Store

Sometimes actors should store large quantities of data and have concurrent access to it. For example, a group of actors process a stream of images and one actor extracts features that need to be used by other actors. These features can be sent via messages to all other actors, but the messaging overhead grows with the number of actors. To prevent this overhead, modern actor-based frameworks use shared atomic objects stores. For example, Erlang offers an Erlang Term Store[15], tables of which can be accessed by different actors. With this approach, the actor only needs to upload features inside a store, while other actors will read them later from the store.

The *E*Actors framework also offers a mechanism to store large volumes of data with concurrent access named the *E*actors Object Store (EOS) API. The EOS API is a set of functions and system actors which can be used to turn a chunk of memory into an objects store. This chunk of memory can be located inside an enclave, and therefore the corresponding object store is private and accessed only by actors located inside the enclave. Alternatively, this chunk of memory can be located in untrusted memory. Accordingly, the corresponding store is public and accessed by all actors. A public EOS can be encrypted or non-encrypted, as well as persistent or non-persistent. Meanwhile, all private EOSs are non-encrypted and non-persistent.

Design

The EOS design inherits the general ideas of the *E*Actors framework. It uses actor's features to provide concurrent access to stored data to different actors located in different enclaves.

---

[15] http://erlang.org/doc/man/ets.html

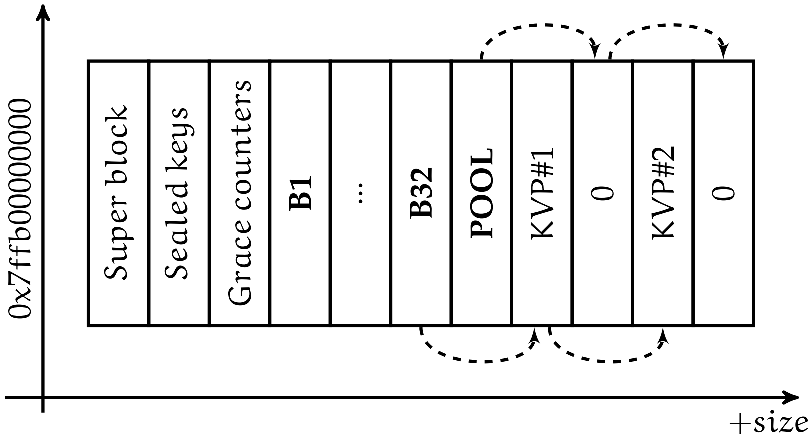


Figure 2.16: Internal structure of an *E*actors Object Store

Figure 2.16 shows the design of a store. In essence, an EOS is a Key-Value Store (KVS). This KVS is split into several *baskets* (B1–B32). The key-value content of the store is distributed over these baskets by hash values of keys. Each basket is an atomic stack[16] and thus, multiple readers and writers can concurrently access the store. Elements of the stacks are nodes filled by the key-value pairs (KVP#1, KVP#2). These nodes are taken from the store node pool (POOL), which is also an atomic stack.

In addition to the store pool and baskets, an EOS has several other components. The first is the *Super block*, which contains information about the size of the store, its version, and some technical information. Secondly, the EOS may include encryption keys, which can be stored in a sealed form inside the dedicated region of the EOS. Thirdly, the EOS includes the *Grace counters*, a data field used in the garbage collection process.

Any chunk of memory[17] can be used as an EOS. For this it needs to be initialised and this process consists of several steps. Firstly, the EOS API initialises internal structures of the future EOS, such as grace counters and baskets. Then the framework converts all free pages of the memory chunk into `nodes`, and pushes them into the EOS' `pool`. After that, the EOS can be used by actors via the *set(k,v)* and the *get(k)* operations provided by the framework.

Set operation

Insertion of data into a KVS by an actor requires three steps. Firstly, the actor needs to obtain an empty `node` of the store. If the store has an empty `node`, it can be

---

[16] Technically, all baskets are implemented as `pools`, i.e. HLE queues with the LIFO semantic

[17] A memory chunk should have enough space for 32 empty nodes and the metadata header

obtained by the *pop* operation applied to the store atomic `pool`. After the successful *pop* operation, the actor gets exclusive access to the `node`. During the second step, the actor fills the `node` with key-value data. In the third step, the actor atomically *pushes* the filled node into one of the EOS's baskets. The number of the basket is identified by the hash function applied to the key of the pair.

Get operation

As can be seen, a basket consists of several versions of key-value pairs (Figure 2.17). However, an actor does not need to 'walk' over the whole stack to find the latest key-value pair. Since baskets of an EOS are atomic stacks, the most recent key-value pairs are always on the top of the baskets, and the first match of keys belongs to the recent key-value pair.

To get a value for the requested key, an actor needs to perform two steps. Firstly, the actor needs to identify the basket which stores the requested key-value pair by computing a hash sum of the key. Secondly, the actor walks from the top (*read top* in the figure) of the basket over all nodes and compares stored keys with the requested key. The first match (*K1V2*) of the key is the requested pair. At the same time, other actors can push new values into this stack (*1* and *2*), but this data is invisible since it lies above the read top.

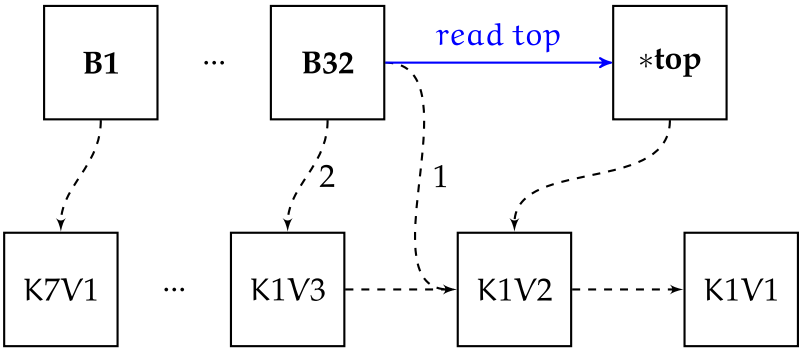


Figure 2.17: Example of a get(K1) operation

This approach has several advantages and disadvantages. The insertion of new values is fast and has constant speed. The retrieval of the stored data depends on the data update rate. Rarely updated data requires more time to retrieve compared to that which is frequently updated. Moreover, since baskets include old versions of key-value pairs, the store can be easily overflowed. Old versions of key-value pairs need to be removed from the store, but this can happen only when these pairs are not accessed by readers. The *Grace counters* and the `Drop list` solve this issue.

Grace counters and Drop lists

One of the important aspects of the use of concurrent data structures is safe removal of outdated elements. An element should not be in use during removal, otherwise, the removal may corrupt the data.

Example: Two $_e$actors use one EOS. The first $_e$actor, named `WRITER`, periodically writes key-value pairs with the same key and different values into the store. The second $_e$actor, named `READER`, reads pairs from the store. The third $_e$actor, named `CLEANER`, needs to remove outdated objects from the store to prevent overflow.

The `READER` actor is accessing a key-value pair. At the same time, the `WRITER` actor has added a new key-value pair into the EOS. Thus, at this moment, two key-value pairs exist inside the EOS, and one of them is garbage, which needs to be removed by the `CLEANER` actor. For this, the `CLEANER` actor (1) needs to identify the old key-value pair and (2) remove it from the basket only when the `READER` has stopped using it.

The simplest solution is to attach all $_e$actors to the same worker. In this case, there is no race condition – all actors are invoked sequentially and there is no situation when the `CLEANER` actor removes an object in use. However, if the actors are attached to different workers, the system becomes concurrent.

There are several approaches to track data in use. For example, some garbage collection algorithms control reference counters [85] to a shared object, and release the object only when no one references the object anymore. Read-Copy Update scheme, implemented in the Linux kernel, separates two phases of shared objects use: the *removal* phase and the *reclamation* phase. The *removal* phase removes an object from a shared structure, while the *reclamation* phase is actual freeing of the object, which happens later, after a *grace period*. The duration of the grace period should be enough to guarantee that all possible concurrent threads have left the critical section, and the removed object is not used anymore.

The framework has its own mechanism of object reclamation, which partly combines both approaches. This mechanism is enabled by two fundamental features of the framework. Firstly, all $_e$`actors` are non-blocking entities. They cannot block their own execution and thus, cannot access a shared object for an unlimited amount of time. Secondly, baskets of the EOS are organised in stacks: the recent key-value pairs are always located on the top of a basket.

Applying this to the previous example: if the fresh key-value pair is inserted into the EOS while the `READER` actor accesses the previous values, another actor will read the recent pair, even if the following read takes place immediately after the end of the previous operation. Consequently, the `CLEANER` actor should neither wait for a grace period (which is impossible, because there is no trusted time source) nor track references to the data (because of the data access pattern). Instead, the `CLEANER` actor just needs to have a list of outdated objects and make sure that all actors involved in interaction with the EOS have made at least one iteration of their own execution.

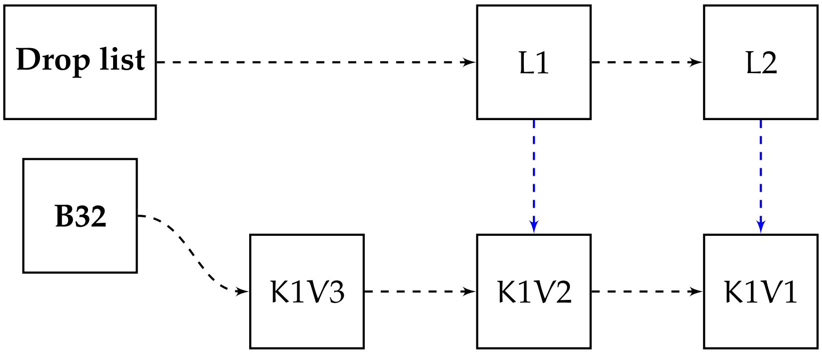


Figure 2.18: Example: Drop list includes pointers to outdated pairs

The process of reclamation of outdated objects consists of three components. Firstly, each actor that accesses the EOS needs to be *registered with the store*. During the registration, each actor receives its own slot inside the EOS's *grace counters*. An actor which deals with the EOS needs to update its own grace counter on each invocation of its body function. By comparing the grace counter's values, one can identify the moment when all actors have processed their own body functions at least once.

Secondly, the *set(k,v)* operation requires special *continuation* after the insertion of a key-value pair. During the continuation, an actor which performed the insertion needs to find a previous element of the basket with the same key. The actor cannot remove this element from the stack since it can be in use. Instead, the actor adds a pointer to this element into the *Drop list* (Figure 2.18).

Thirdly, a system actor named `CLEANER` periodically checks the *Drop list*. If the `CLEANER` actor detects a new pointer in the list (`L2`, for example), it waits until all counters inside the grace list change their values, then removes the referenced element (`K1V1`) from the basket (`B32`), and then removes the corresponding pointer (`L2`) from the list.

The framework does not guarantee that the `CLEANER` actor removes objects faster than new objects appear. However, the design of the framework enables a delay of inserts: after an unsuccessful insertion attempt, the actor can repeat the attempt during the next execution iteration. Moreover, in accordance with the store role, the Drop list can be disabled, as can the `CLEANER` actor and grace counters.

Encryption of objects

The EOS API supports encryption of key-value pairs. The encryption key is stored in a private compartment of actors and disseminated via encrypted cargos. The choice of baskets in encrypted EOSs is made by a hash value of the deterministically encrypted key. To retrieve an encrypted key-value pair, an actor does not decrypt all stored objects. Instead, it encrypts the requested key and performs a search in baskets by comparison of the encrypted key with stored objects. Additionally, to preserve the integrity of the pairs, the EOS API does not store the keys and the values separately, but stores the encrypted pairs as combined values.

Persistence on demand

The design of the EOS API allows a developer to create a persistent public EOS, i.e. a store, objects of which survive a system restart. For this, the framework can use a memory mapped file instead of allocation of virtual memory for an EOS. An EOS located inside this memory region becomes persistent without the use of the input/output system actors or ECALLs.

Two features of a system make this approach possible. Firstly, enclaved software can access not only the enclave's address space but the whole virtual memory range of the host process except the virtual addresses of other enclaves (section 2.1). Secondly, interaction with secondary storage in the Linux kernel is implemented via the page cache [86], pages of which can be mapped into a process address space via the `mmap` system call [87]. Thus, one can map a file or a storage device to a virtual address and then, an enclaved software can modify the storage content in the same way, as it can modify any memory object located inside untrusted memory. The mechanisms for mapping of files (`mmap()`, `munmap()`) and data syncing (`fsync()`) are provided by the corresponding system actors.

The framework does not protect persistent EOSs against reboot and fork attacks. However, this could be addressed by adopting an approach such as LCM [88] or ROTE [89].

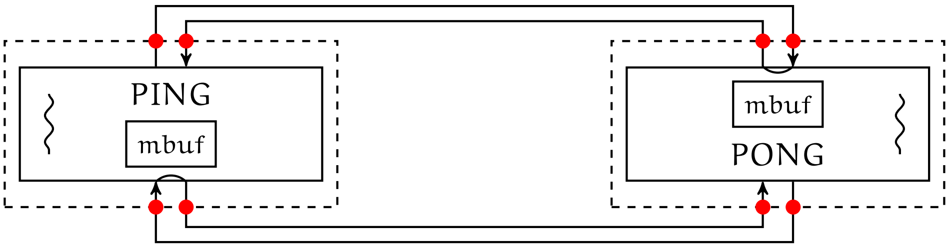
2.2.3 Microbenchmark and use cases

One microbenchmark and two use cases were implemented on top of the *E*Actors framework: (i) a microbenchmark of inter-enclave communication, (ii) a secure multi-party computation (SMC) service, and (iii) a secure instant messaging service. All feature the use of multiple enclaves and differently exercise the core components of the framework. The microbenchmark compares the message-based communication interface of the framework with the ECALL interface provided by the SGX SDK. The SMC service demonstrates how mutually-distrusting parties can securely perform computations without revealing the individual secret values. Additionally, this service benchmarks encrypted cargos. The secure instant messaging service, in turn, demonstrates how the framework can be used to partition an application which processes data of clients. This service actively uses system actors and the EOS API.

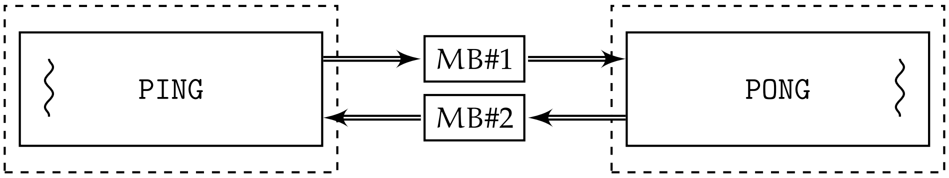
2.2.3.1 Microbenchmarking inter-enclave communication

The *pingpong* microbenchmark consists of `PING` and `PONG` actors. The `PING` sends a message to the `PONG`, and the `PONG` replies to the `PING` with a message. Two variants of *pingpong* were compared: an *E*Actor-based implementation and a native SGX SDK-based approach. Figure 2.19 depicts differences in these implementations.

As can be seen in the figure, in the native SGX SDK-based scenario, the `PING` and the `PONG` components are located inside different enclaves. To send a message, the `PING` issues the OCALL to leave its own enclave, and then issues the ECALL to enter into the enclave of the `PONG`. After the entry, the `PING` needs to deliver a message, and then leave the `PONG` enclave and re-enter its own enclave. The `PONG`, at the same time, after the receiving the message, repeats the same sequence of actions and sends the answer to the



(a) Native SGX SDK



(b) Cargos

Figure 2.19: Design of microbenchmark scenarios

`PING`. Because both actors have their own contexts (pthreads), the data exchange needs to be protected. A simple spin-lock [90] was used for the data access synchronisations.

In the framework-based scenario, the `PING` and `PONG` are designed as two $_e$actors located inside two different enclaves. Here, $_e$actors's threads are bound to different CPU cores, and non-encrypted `mboxes` are used.

2.2.3.2 Secure multi-party computation service

Several protocols which enforce secure multi-party computations are provided in the literature. A secure sum protocol [91, 92] has been chosen for the framework evaluation. This protocol aims at securely computing the sum of all the inputs of a set of participants without revealing the individual values.

Protocol description

The original protocol was slightly improved for the use case. Firstly, the original protocol targets a distributed setting where the individual participants exchange messages over the network. With the support of trusted execution, all participants can be represented by enclaves which are co-located on a single machine. This way, the costly network-based communication between the participants can be avoided. Secondly, the use case generalises the original protocol by performing the sum of private vectors instead of individual values.

Figure 2.20 shows the general scheme of a secure-sum service. In this figure, `k` parties (i.e., `P1`, `P2`,..,`PK`), are connected to each other in a ring structure. Each party

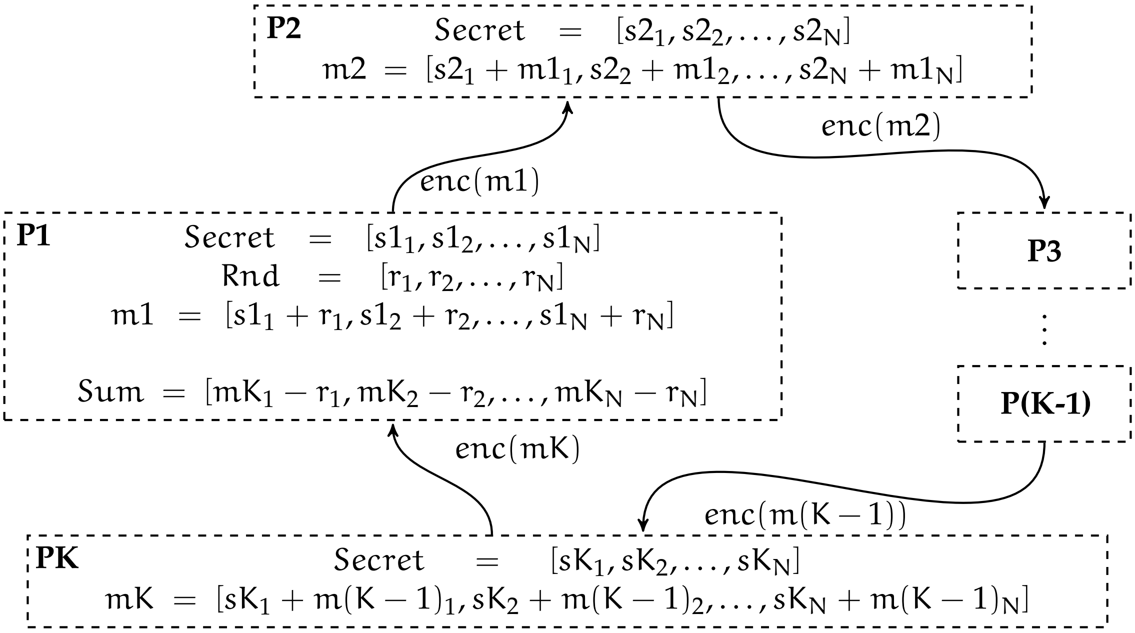
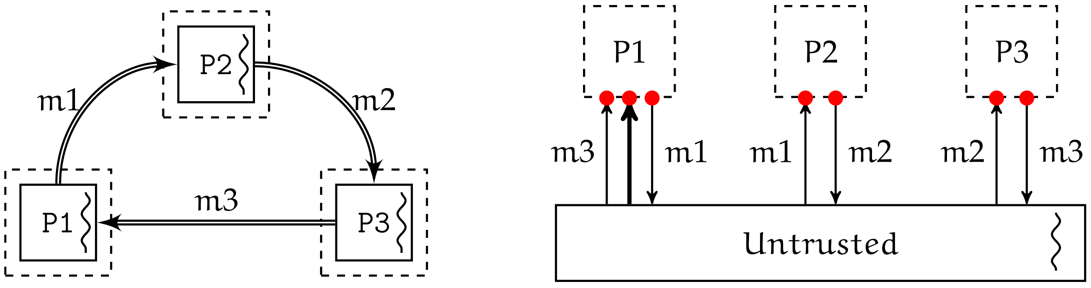


Figure 2.20: General scheme of the SGX-based secure multi-party sum

has its own enclave and stores a secret input vector (`Secret`). On demand, the SMC scheme computes the sum of the secret vectors. To do that, the first party `P1` starts by generating a vector of random values `Rnd` of the same size as the secret vector. Then, `P1` generates a message vector `m1`, which is the sum of `Rnd` and the secret vector `Secret`. After encryption, this message is delivered to the second party `P2`. The second party decrypts the message, adds it to its own secret vector, encrypts the resulting message and sends it to the next party in the ring. This process is repeated until the last party `PK` delivers the `mK` message to the first party `P1`. Finally, the `P1` computes the result of the sum by subtracting the `Rnd` vector from the latest received vector `mK`. This result is then shared among all the participants.

Design

To highlight the benefits of the *E*Actors framework, two different variants of this protocol were designed (Figure 2.21. The left part of the figure shows the SMC use case implemented with *E*Actors while the right part shows the same use case implemented with the SGX SDK. In the first case, each party is implemented as an independent actor with its own worker and an SGX enclave image. In the second case, each party is also implemented as an SGX enclave but only a single thread executes the protocol by entering and leaving one enclave after another.



(a) *E*Actor-based deployment (b) SGX SDK-based deployment

Figure 2.21: Different implementations of the secure sum protocol

The threat model of the SMC sum protocol is based on an assumption that all parties are mutually-distrusted. Thus, a malicious party or an adversary would try to listen to the network or messages to obtain the temporary sum or guess any secret value. Encrypted connectors prevent this, since each pair of connectors have its own encryption key stored inside an enclave, and parties are attested during the key exchange procedure (see section 2.2.2.1). The former prevents the leakage of intermediate sum values while the latter guarantees that only expected parties are involved in the SMC sum.

2.2.3.3 XMPP instant messaging service

Instant messaging is used to exchange privacy-sensitive information. Trusted execution builds a means to make it more secure. Accordingly, an *E*Actors-based variant of an instant messaging service that exercises all the outlined features were designed and implemented. Besides security based on the use of multiple enclaves, performance and scalability were prime design goals.

The developed instant messaging service implements core parts of the Extensible Messaging and Presence Protocol (XMPP) protocol [93] and supports two types of communication: One-to-One (O2O) and One-to-Many (O2M). The O2O type allows end-to-end encrypted messaging between two participants which resembles the *de facto* approach for modern messengers [94]. In principle, this type of connection can be managed inside a single enclave as only the information regarding online users and statistics has to be secure. For the One-to-Many (O2M) type, the situation is different as it offers support for group chats. Here, the server decrypts the messages of each user and re-encrypts for each member of the group. While a single enclave could also be

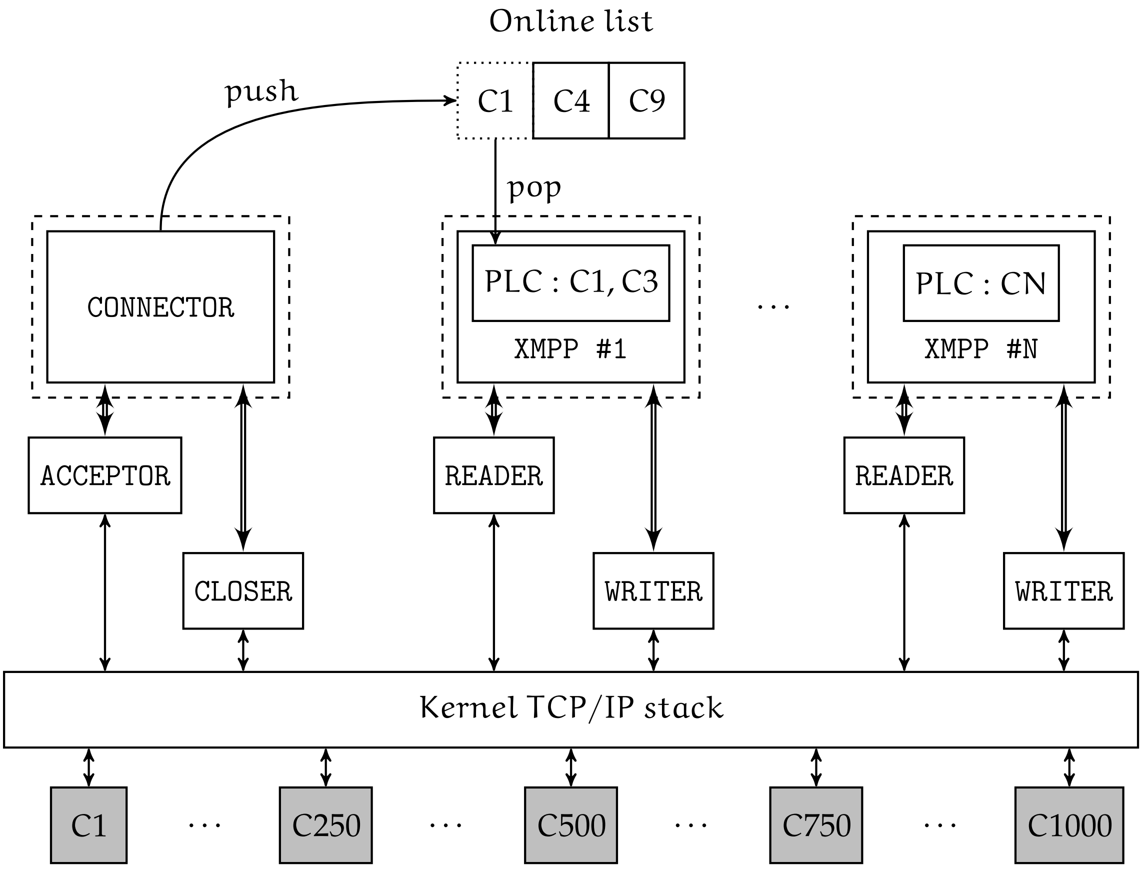


Figure 2.22: XMPP service architecture

used in principle, the implementation supports a dedicated enclave for each group chat to improve isolation.

Architecture

Figure 2.22 shows the architecture of the XMPP service. The service includes various components: system actors, the set of XMPP actors, a queue for data exchange between actors, and more. Below, each component is considered independently.

Actors The XMPP service includes three groups of actors. Firstly, there are several system actors (section 2.2.2.2). The service uses `ACCEPTOR` and `CLOSER` actors. The former, at the construction phase, opens a TCP/IP socket[18], and then during the active execution, accepts incoming connections (`accept()`) by requests. The latter is used to close a connection. Additionally, the service spawns multiple `READER` and `WRITER` actors

---

[18] Calls `socket()`, `bind()`, and `listen()` system calls

in accordance with performance needs. These system actors are used to receive and send data via a TCP/IP socket.

Secondly, there is a `CONNECTOR` actor, the main function of which is the connections control. This actor periodically sends requests to the `ACCEPTOR` actor to accept incoming connections. After acceptance of a connection, the `CONNECTOR` actor adds a description of this connection into the *Online list*. The *Online list* is a `pool` with one writer (`CONNECTOR`) and multiple readers (`XMPP` actors).

Thirdly, there is an `XMPP` actor, which implements an `XMPP` server function. This actor interacts with clients, obtained from the *Online list*, by sending requests to `READER` and `WRITER` actors. It is the 'heart' of the service since it implements communication between `XMPP` clients. Similarly to `READER` and `WRITER` actors, the service can be configured to spawn multiple `XMPP` actors to increase the performance of the service.

Application specific components The service includes several application-specific components. The first is the *Online list* mentioned above. This list is located inside untrusted memory and shared by the `CONNECTOR` actor and all `XMPP` actors. The `CONNECTOR` actor pushes the descriptors of incoming connections into this list. Idle XMPP actors, in turn, concurrently pop the descriptors one-by-one and process them inside their own enclaves.

The second component is the Private List of Clients (PLC). Each `XMPP` actor interacts with clients, which are described by the *client abstractions* stored inside a PLC. A client abstraction includes various elements, for example a TCP/IP socket of the client, the current state of the client in accordance with the XMPP client model, an encryption context, and more.

Additionally, there is the EOS, which is used by all XMPP actors (not depicted in Figure 2.22). It stores information about currently opened connections like the binding of a particular JabberID (JID) and the corresponding network socket.

Resource assignment The minimal configuration of the XMPP server includes: one `CONNECTOR` actor with a pair of `ACCEPTOR` and `CLOSER` actors, and one `XMPP` actor with a pair of `READER` and `WRITER` actors. For simplicity, the first group is named the *CAC group*, while the second one is named the *XRW group*. The XMPP service scales horizontally by increasing the number of XRW groups.

Key features of the design

The XMPP service is designed in accordance with a major goal – scalability. The service should be horizontally scaleable, i.e. the performance of the system should grow while increasing the number of `XMPP` actors. The following paragraphs describe how this goal was reflected in the design of the XMPP service.

Scalability By design, the network actors are stateless and symmetric. Any network actor can be used by any `XMPP` actors. Thus, one, two, or any number of `XMPP` actors can use any number of network actors. The microbenchmark was developed to estimate the performance of the services with different configurations of actors. It demonstrated that the maximum performance was achieved when a single `XMPP` actor is used together with two `READER` and `WRITER` actors. Moreover, the `XMPP` actor should use its own worker, while the `READER` and `WRITER` actors should share a single worker. In sum, the overall design of the service is based on the idea that the horizontal scalability is achieved by increasing the number of `XMPP` actors, which are coupled with `READER` and `WRITER` actors.

Information about connections All incoming connections are dynamically assigned to XMPP actors. These XMPP actors can be located in different enclaves to ensure isolation. While XMPP actors are independent, they still need to have the ability to exchange data between each other.

Example: There are two clients which are connected to the same XMPP server. These clients are served by different `XMPP` actors located inside different enclaves. Each `XMPP` actor has a client abstraction of the corresponding client. This client abstraction is stored inside a PLC, and includes information about the JID and the TCP/IP socket of the client. However, to send a message from the first to the second client, the actor which serves the first client needs to have access to the data stored in the enclave of the second XMPP actor, which is impossible.

Shared access to connection information from different enclaves is provided by an EOS. In a case of multi-enclave setup, this EOS is located inside untrusted memory and stores encrypted data. Enclaved `XMPP` actors have concurrent access to this data and do not require any synchronisation mechanisms or complex message exchange protocol.

In the case of a single enclave setup, this EOS is located inside the enclave and stores data non-encrypted. Each group chat also has its own private non-encrypted EOS.

Distribution of clients By design, there is no assumption about the equal distribution of the clients processed by different XRW groups. As mentioned previously, the `CONNECTOR` actor accepts incoming connections and adds information about them inside the *Online list* in the form of the *client abstraction*. Only an idle `XMPP` actor can pop a client from this list and thus, a situation when one `XMPP` actor has more clients inside own PLC then other is possible. This disproportion does not play a role from the performance point of view, because the equal distribution of clients does not mean equal distribution of network load. However, from the security point of view, this disproportion can be considered as a vulnerability, and as a consequence, each `XMPP` actor had a pre-defined limit[19] of processing connections.

Operation

At the beginning, only the `CAC` group works actively. The `CONNECTOR` actor periodically requests the `ACCEPTOR` actor to accept incoming connections. On success, the `CONNECTOR` actor prepares a new record for the *Online list*. The `XRW` groups are working at the same time, but all actors stay in the *idle* state since no one of them has clients in PLCs. Periodically, the `XMPP` actors try to dequeue a connection from the *Online list* and on success, the `XMPP` actor changes its state to *active* and adds this task to its PLC.

In the active state, all `XMPP` actors perform the same activities. Each `XMPP` actor, firstly, goes over all its client abstractions, which are located inside the PLC. From each abstraction, the actor retrieves the connection socket descriptor, and pointers to the `mbox` and to the `pool` belonging to this abstraction. Then, the actor combines this data into a batched request and sends it to its `READER` actor.

A `READER` actor receives a request from own `XMPP` actor, comes across the list of triplets (socket, `pool`, `mbox`) and performs the non-blocking *read* system call on each socket. If a socket has an incoming message, it will be read into a `node` dequeued from the `pool` and then sent to the `mbox`. In addition, the `READER` actor generates and sends back a special bitfield mask. This describes which client abstractions from the read request received an incoming message.

---

[19] The maximum number of clients divided by the number of `XMPP` actors

After the receiving of an answer, an `XMPP` actor checks the bitfield mask from the answer, and then interacts with each client abstraction which received network data. The `XMPP` actor processes client's incoming data in accordance with the XMPP specification and the current state of the client. After successful dispatching of incoming messages, the `XMPP` actor changes its state, tries to add a new client abstraction into its PLC, and then prepares a new batched read request again.

Name resolving When an `XMPP` actor receives a new client abstraction, this abstraction consists of three elements only: a network socket descriptor, a `pool`, and a `mbox`. At this moment, the actor has no information that the client is attached to the socket. This information becomes available only after performing the authentication when a client sends its JID to the server. After that moment, the actor has the full information about the client. The `XMPP` actor pushes the binding of JID to socket into an EOS, and other actors which need to send a message to this JID can retrieve it from the EOS.

Message delivery When an O2O `XMPP` actor receives a message from a client, it firstly checks necessary headers in accordance with the XMPP specification. Secondly, it retrieves the JID of the recipient from the message. Thirdly, it retrieves the recipient's TCP/IP socket from the EOS. Then, if the recipient is online and in the proper state (this information is also stored inside the EOS), the actor asks its `WRITER` actor to send a message through the retrieved TCP/IP socket to the recipient.

An O2M `XMPP` actor behaves slightly differently. Each O2M `XMPP` actor has a list of clients, which share the same chat room. When an actor receives a message, it retrieves the room identifier, and after necessary checks, sends the message to all participants from the list step-by-step.

Disconnect of a client If a client drops the connection, the XMPP engine detects this by the reading of an error from the client's TCP/IP socket. The bitfield mask returned to a `XMPP` actor reflects this situation. Later, the `XMPP` actor purges the state of the corresponding client inside the EOS and then removes the client abstraction from its PLC. After that, the dropped client can reconnect to the server again, and this reconnection will be processed again by the `CAC` group.

Client support The *burster* utility has been developed for evaluation purposes. This utility was written in C language, used the *libstrophe* [95] library and provided the basic functions of an XMPP client: connection to a server, authorisation, sending and receiving of messages. The utility emulated the behaviour of multiple concurrent clients by the spawning of multiple threads, each of which implemented an independent XMPP client.

2.2.4 Evaluation

This section demonstrates evaluation of the microbenchmark and two services described previously.

For the evaluation, two servers based on Intel Xeon CPU E3-1230v5 (3.40 GHz, 4 cores, 8 hyper-threads) were used. Both of them were equipped with 32 GiB of RAM and Mellanox MT27520 RoCE RDMA controller (10 GbE, RDMA capabilities were not used). Both servers had equal software systems, based on Ubuntu 16.04.3 with the Linux kernel version 4.4.0-109 and the Intel SDK version 1.8 with builtin Intel IPP library. All modules were built with the ”-O2” optimisation flag.

The framework contains roughly 6200 lines of C code. The part of the framework embedded in an enclave contains 3278 lines of code and some of the third-party libraries shipped with the SGX SDK. As a result, for an application such as the XMPP server, each enclave was limited by 512 KiB of memory.

2.2.4.1 Inter-enclave communication

The pingpong benchmark measures (i) the performance of inter-enclave communication implemented by different approaches, and (ii) the impact of message sizes on the performance. For each communication scheme (mbox and ECALLs), and for each message size (from 16 B to 512 KiB), the time of 1.000.000 ping-pong message exchanges was measured. These measurements were repeated five times for each configuration and then, after the averaging, the throughput was computed in MiB per second by dividing the transferred data size by the measured time. All messages within the same test had the same length and pseudo-random content. The transfer time included not only the pure data transfer time but also the payload generation time.

Figure 2.23 clearly show that *E*Actors (`EA`) outperforms the native SGX SDK (`Native`). As can be seen, the native SGX SDK reaches its peak throughput near 32 KiB.

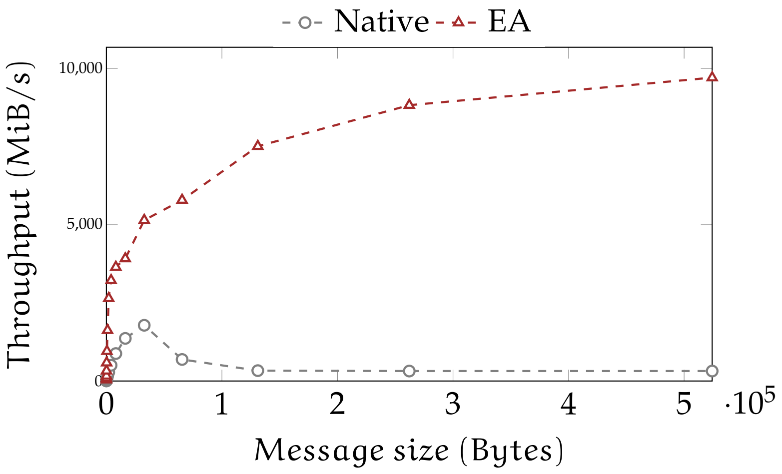


Figure 2.23: Throughput of different communication interfaces

This is explained by the fact that for each OCALL, the SDK allocates a memory space in which the sent message is copied. However, after reaching the L1 data cache size, which is 32 KiB in Intel Skylake Core [96], memory copy becomes slow. The minimal difference between `EA` and `Native` is approximately 2.8× obtained for 16 KiB messages. For the small messages (smaller than 64 bytes), the difference is approximately 16.8×, while for larger messages (longer than 512 KiB) the difference exceeds 30×.

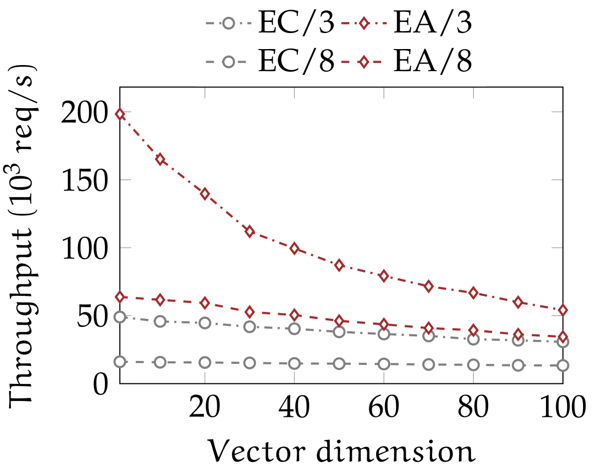
2.2.4.2 Secure multi-party computation service

The two deployment s of the secure multi-party communication service depicted in Figure 2.21 were implemented: an *EActors*-based implementation and a SGX SDK-based version. For a predefined number of parties and vector dimensions, 10.000 invocations of the secure sum were generated and the response time was measured. These measurements were repeated at least three times for each configuration and then, after the averaging, the throughput was computed.

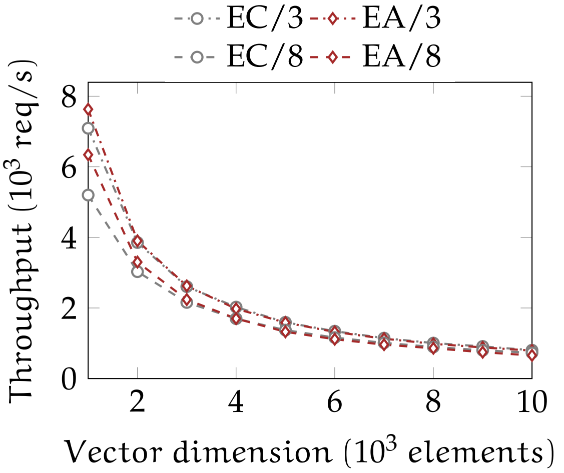
The measured throughput depends on multiple factors, such as the speed of encryption/decryption of messages, number of parties, vector length and the duration for generating random numbers. The last factor is crucial because the first party needs to refill the `Rnd` vector on each request.

Performance of the plain protocol

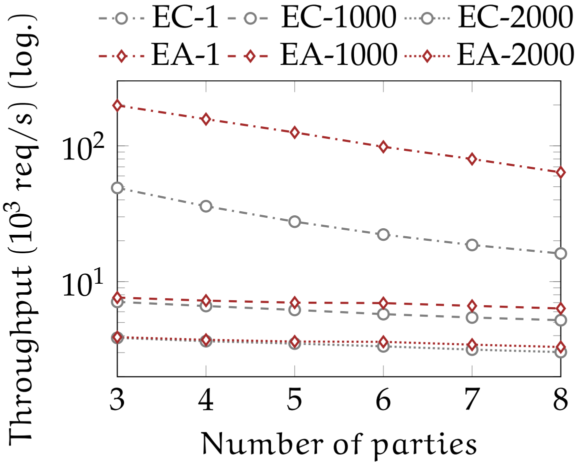
In the first experiment, the plain execution of the protocol was considered. Figure 2.24a and Figure 2.24b show the performance of the SMC service for short (below



(a) Throughput for short vectors

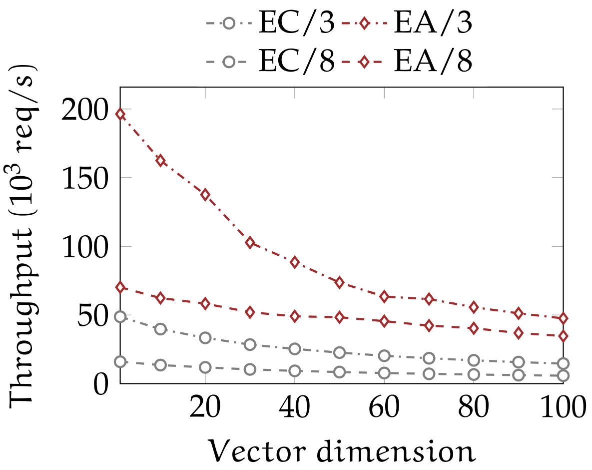


(b) Throughput for long vectors

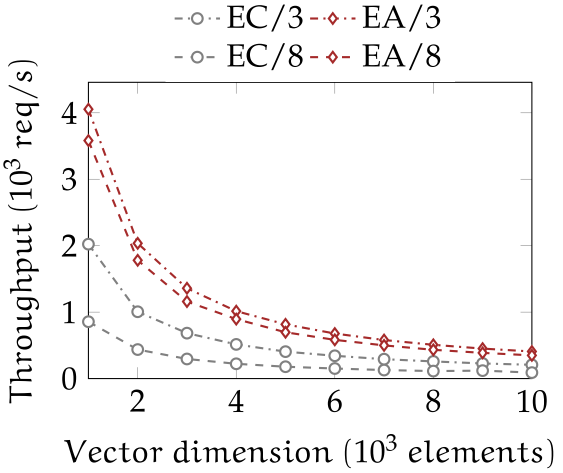


(c) Impact of number of parties

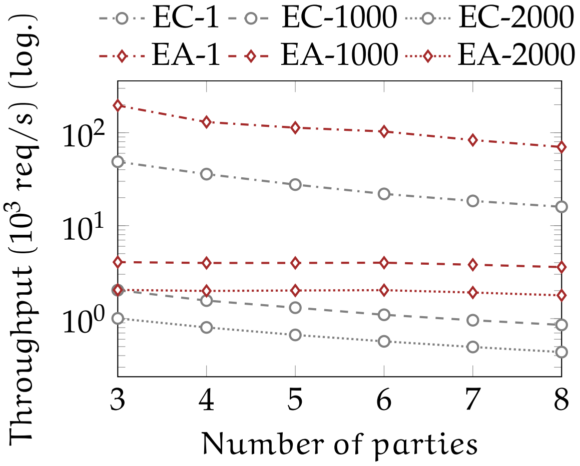
Figure 2.24: SMC scheme with the plain protocol



(a) Throughput for short vectors



(b) Throughput for long vectors



(c) Impact of number of parties

Figure 2.25: SMC scheme with dynamically computed input vectors

100 elements) and long (between 1000 and 10.000 elements) input vectors. The two ″extreme″ configurations were used: three and eight[20] participating parties. Figure 2.24c shows the performance impact depending on the number of parties. For this benchmark, three different vector sizes were used: 1, 1000, and 2000 elements.

Firstly, as can be seen, the throughput of the *E*Actors-based implementation is higher than the throughput of the SDK-based implementation, especially for short vectors. Increasing the vector length leads to the degradation of the throughput. The same applies to increasing the number of parties. However, different implementations degrade differently with an increasing number of participants. As seen in Figure 2.24a and Figure 2.24c, for the same length of vectors (1), the difference in the throughput between three parties (EA/3 versus EC/3) is 3.65×. In the case of eight parties, the same metric reaches 3.96×.

Secondly, as shown in Figure 2.24b, for long vectors, the difference between the two implementations is not so severe as with short vectors. For example, the difference in throughput for three parties and 1000 elements (EC/3, EA/3) is 8%, and it becomes negligible for vectors longer than ≈2000 elements. However, the number of parties is still an important factor. For eight parties, the difference in throughput is 22% for 1000 elements, which becomes negligible for vectors longer than ≈4000 elements.

To explain this behaviour, three possible sources of performance degradation were identified. The first one is transition costs of entering and leaving trusted execution mode during ECALL/OCALL use. The implementation of the SDK-based SMC scheme uses ECALLs efficiently, i.e. transition costs do not involve copying of memory, and thus, the transition costs do not depend on the vector length. However, the number of parties increases transition costs proportionally.

The second possible source of performance degradation is encryption and decryption of messages, since the vector size impacts the encryption/decryption demand linearly. To identify this impact, the *pingpong* microbenchmark was modified to use encrypted cargos. The encrypted *pingpong* application reached the throughput of 1 GiB/s, which is roughly 35 times higher than the throughput of the EA/3

---

[20] The minimum number of parties in the SMC sum and a number of hyper-threads in the CPU

configuration for 1000 elements ($7092 * 1000 * \texttt{sizeof}(\texttt{uint32\_t}) \approx 27\ \texttt{MiB/s}$), thus connectors bandwidth is not the limiting factor.

A detailed analysis revealed that the source of the performance degradation is a slow `sgx_read_rand()` SGX SDK function. In accordance with the protocol, the first party needs to generate a vector of random values before each request. An increase of the vector length leads to increased usage of the trusted random number generator.

Performance with dynamically computed input vectors

In the previous evaluation, it was assumed that the parties do not perform any computation beside the bare protocol. In this case, the parties update their internal secrets after each computation of the secure sum. For this, each party generates a random vector and adds it with the secret one. Figure 2.25 shows the performance of the two SMC systems with such an additional workload applied.

As can be seen, this additional computation significantly impacts the performance. For example, for a vector size of only one, the performance benefit for *E*Actors has grown to 4× for three parties, and to 4.4× for eight parties (Figure 2.25b). For longer vectors, the difference grows faster. For example, for 2000 elements (Figure 2.25c), the difference in throughput grows from 2× (three parties) to 4.1× (eight parties). Moreover, in contrast to the plain execution of the protocol, experiments with very long vectors also show a significant difference in throughput. Figure 2.25b demonstrates that for eight parties and any vector length in the tested range, the *E*Actors-based implementation is at least 3.88× faster.

This experiment clearly shows that additional computations, which need to be performed by parties, can be and should be parallelised by $_{e}\texttt{actors}$. In contrast, if parties do not need to perform computations between invocations, they can be implemented directly on top of the SGX SDK.

2.2.4.3 XMPP instant message service

The evaluation of XMPP service compares the performance of the XMPP service built on top of the framework and the performance of two popular non-enclaved XMPP servers: JabberD2 and ejabberd. JabberD2 is written in C, has multiprocess design and consists of several services, each of which implements a unique function, like routing of messages between different servers or interaction of clients with the server. Moreover, the JabberD2 server supports Secure Sockets Layer (SSL) traffic encryption and group chats by the MU-Conference module. ejabberd is written in Erlang and has an actor-based execution model and also supports traffic encryption and group chats.

The number of messages processed by a service was used as the performance characteristic. Despite the comparison to JabberD2, ejabberd and *E*Actor-based XMPP services, the several aspects of the actor-based system were additionally considered:

1. Performance of a single XMPP actor for both communication schemes: One-to-One (O2O) and One-to-Many (O2M)
2. Scalability of the service
3. Impact of the number of enclaves and the enclaving on the performance

These aspects were evaluated in various configurations and combinations:

- Tested number of XRW groups: 1–32
- Number of concurrent clients: 20–1000 (O2O), 10–100 (O2M)
- CPU and enclave affinities: 1–32 enclaves, 1–8 hyperthreads

Behaviour of clients

The *burster* utility, introduced in section 2.2.3.3, was used for the load generation. This utility enables spawning of multiple clients, each of which represents a full-fledged client which can connect to a server and exchange messages.

The burster was configured to implement the following sequence of actions. At the start, the burster connects all clients one-by-one to the server. During this operation, the burster assigns *ping* and *pong* roles to the clients in accordance with the evaluation scheme. In the case of the O2O evaluation, the first half of clients receive the *ping* role, the second half – the *pong* role. In the case of the O2M evaluation, the first client receives the *ping* role, while the remaining get the *pong* role. After the successful connection and authorisation of the last client, the burster initialises a timer and releases a global lock, which enables the active behaviour of all clients.

During the active work, each client with the *ping* role sends a message to the server. These messages have the necessary attributes in accordance with the XMPP specification, encrypted (in the O2M case), and have different bodies for the *ping* and

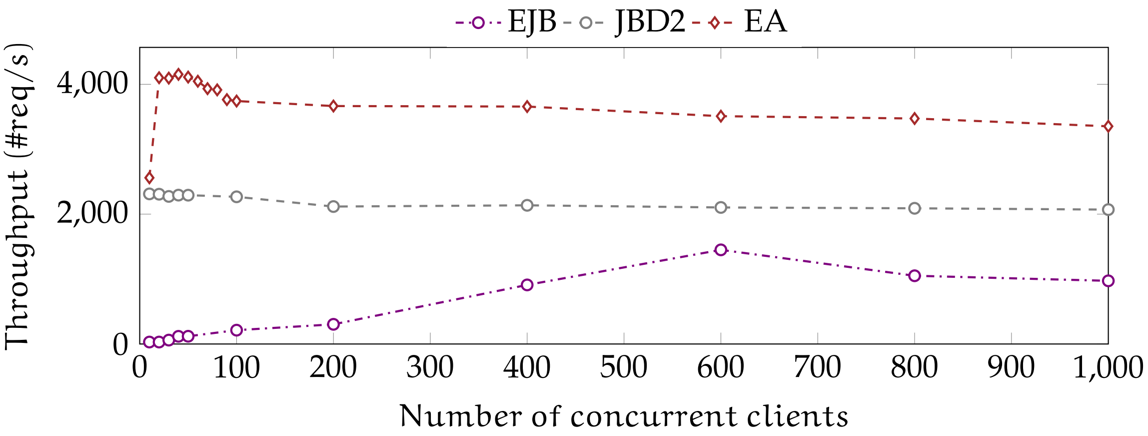


Figure 2.26: Single XMPP actor performance, O2O mode

pong messages[21]. In the case of the O2O mode, the messages include the JID of a random *pong* client, while in the case of the O2M mode, these messages include the name of a group chat which includes all *pong* JIDs. A *pong* client in O2O mode sends an answer to a *ping* client. After receiving the answer, the *ping* client increments a performance counter and sends a new message. In the case of the O2M mode, the group chat server sends messages to all *pong* clients, and then notifies the *ping* client that all messages have been delivered. After that, the *ping* client increments the performance counter and sends the next message.

After the end of the pre-defined time interval, the burster stops the execution of the clients and accumulates all performance counters, i.e. number of messages processed by each *ping* client.

Single XMPP actor performance

Figure 2.26 and Figure 2.27 show a performance comparison of ejabberd (`EJB`), JabberD2 (`JBD2`) and the framework-based solution (`EA`). The `JBD2` used the default set of components, such as c2c, s2m, router and others for the O2O communication scheme, and the additional service named MU-Conference in the O2M case. The `EJB` also used a default configuration. The `EA` framework-based service consisted of the single CAC and the single XRW groups[22].

---

[21] The overall size of the messages did not exceed 150 bytes

[22] As mentioned previously, one CAC and one XRW groups use two enclaves: one for the `CONNECTOR` actor in the CAC group, and one for the `XMPP` actor of the XRW group

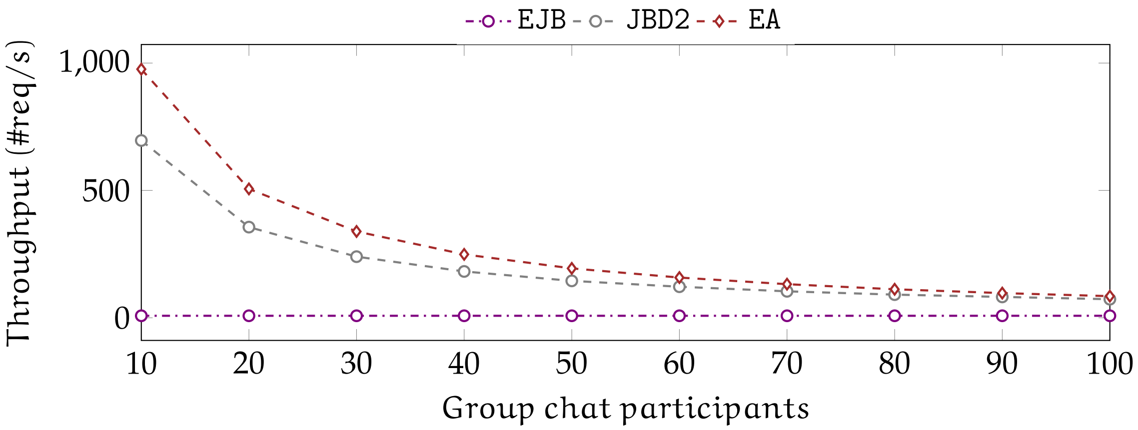


Figure 2.27: Single XMPP actor performance, O2M mode

As can be seen in Figure 2.26, the `EA` significantly outperforms the `EJB` and `JBD2`. However, all services behave slightly differently. The performance of the `EA` service grows rapidly at the beginning, then has a performance "pike", falls significantly and then degrades slightly with the number of clients. The performance of the `EJB` grows very slowly until the number of clients does not reach 600, and then slightly degrades. The performance of the `JBD2` service, in turn, has the performance maximum at the beginning, and then, after the fall near the 150 clients, degrades very slowly with the number of clients (1% over 400 clients). In the "pike" of the `EA` (40 clients), the `EA` service shows 1.81× more throughput than the `JBD2` and 35.1× more throughput than the `EJB`. For 600 clients, when the `EJB` has the maximum performance, the `EA` outperforms the `JBD2` by 1.67× and the `EJB` by 2.42×. For the 1000 clients, these differences reach 1.62× (`JBD2`) and 3.45× (`EJB`).

The mechanism of dispatching TCP/IP packets defines the difference in the behaviours. The network components of the JabberD2 server interact with the client's sockets directly and can process the messages immediately when they arrive. Thus, JabberD2 does not need to read sockets manually. Instead, it can use the *select* system call for waiting for incoming messages. The EA service, in turn, cannot do the same. Firstly, the actors cannot be blocked by the *select* system call because they can be attached to workers which provide the execution contexts to other actors. Secondly, the actors do not store information about the connections nor do they have encryption keys of the connections since they are stored inside enclaves. As a result, to receive incoming messages, the trusted actors periodically send requests to network actors - that process degrades with the number of clients. As many clients are connected, more time is needed to prepare requests for network actors. However, this feature can be compensated by increasing the number of XRW groups.

Figure 2.27 compares the throughput of the same services but in the group chat mode. In contrast to the O2O scheme, the impact of the request-based networking is negligible for this number of clients and the `JBD2` and the `EA` services scale in the same way with an increasing number of clients, while the `EJB` service shows a constant throughput. The throughput of the `EA` is 1.40× higher than the performance of the `JBD2` for 10 clients, and more than 100× compared to the `EJB`. Within the number of clients, the difference between the `JBD2` and the `EA` decreases down to 16% for 100 clients and becomes indistinguishable for more than 150 clients.

Scalability of the XMPP service

Several characteristics of the framework-based XMPP service were identified during the evaluation. Firstly, a single `XMPP` actor shows the maximum performance with a single `READER` actor and a single `WRITER` actor. Increasing the number of `READER` and `WRITER` actors, as well as increasing the number of `XMPP` actors attached to a single pair of `READER` and `WRITER` actors does not increase the performance of the group.

Secondly, the framework-based XMPP service scales linearly with a number of XRW groups. The CPU affinity also does not play a role: the system showed the same performance for the same number of XRW groups assigned to the first CPU core, or evenly distributed over all cores. The CPU affinities of the XRW groups becomes important only when the number of XRW groups per physical CPU core reaches a special threshold.

Figure 2.28 shows the performance of the single enclave O2O XMPP service with different configurations of actors: a single XRW group (`X1`), two XRW groups (`X2`), and 16 XRW groups (`X16`). As can be seen, the `X2` and the `X16` repeat the behaviour of the `X1`. They grow rapidly at the beginning, then reach the extremum value, and then start degrading slightly. The extremum value grows with the number of workers. For a single XRW group, the extremum is 40 clients, for two XRW groups this value is 50, while 16 XRW groups show the maximum throughput with 400 clients.

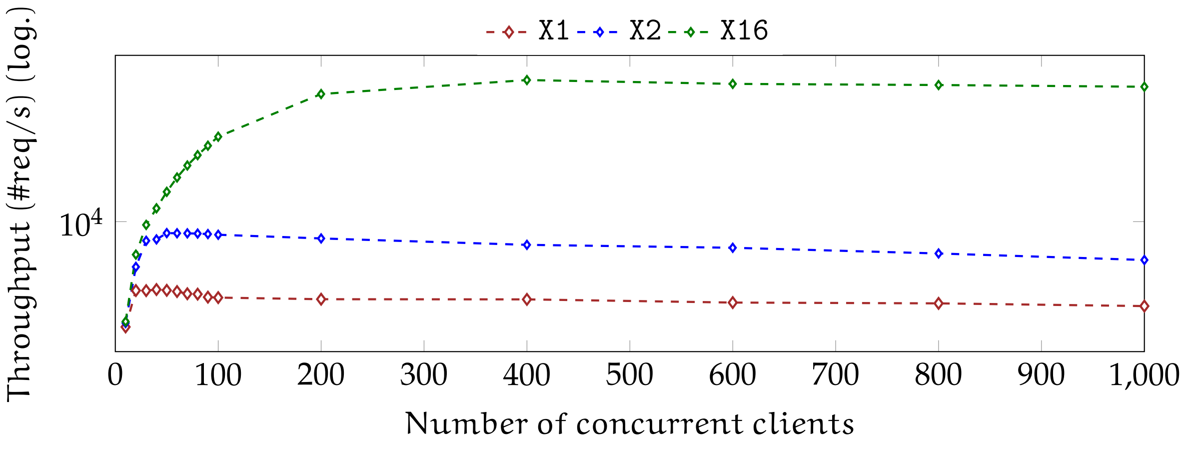


Figure 2.28: Scalability with different number of $_e$actors, O2O mode

The throughput of two XRW groups in the peak is 2.09× higher than the performance of a single XRW group for the same number of clients. 16 XRW groups at the peak demonstrate 8.43× higher performance than 2 XRW groups, and 17× higher than the performance of a single XRW group.

16 is the maximum value of XRW groups that can be assigned to one physical CPU core (two hyper-threads) on the evaluating hardware. More than 16 XRW groups do not increase the performance since the system becomes overloaded. The maximum observed performance per single CPU core is 62471 requests per second. Additional XRW groups can be assigned to other CPU cores, and the maximum possible performance of the evaluating platform can be estimated at ≈250000 requests per second.

Impact of number of enclaves and enclaving

Two additional aspects of enclaves were evaluated: how the number of enclaves impacts on the performance of a system, and how the enclaving of actors impacts on the performance. For that, two experiments were performed, both of which were based on the O2O communication scheme of the framework-based XMPP service.

During the first experiment, the impact of enclaves was estimated. 16 XRW groups were evenly assigned to several enclaves and the performance of the service with 400 clients was measured. This experiment was repeated several times for the following number of enclaves: 1 enclave (16 XRW groups inside), 2 enclaves (8 XRW groups per each) and 16 enclaves (1 XRW group per each). Figure 2.29 shows the averaged results.

As can be seen, there is no difference in the performance for more than one enclave: both 2 and 16 enclaves show the same performance values. However, there is a

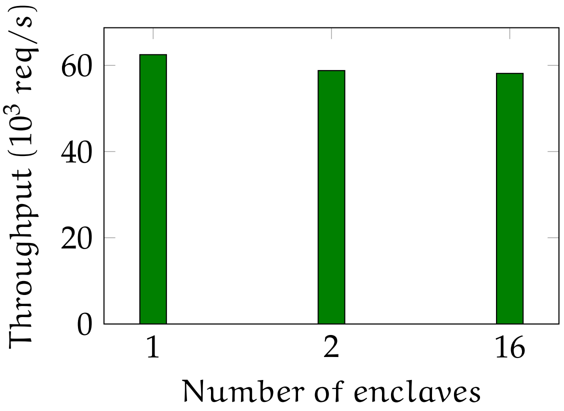


Figure 2.29: Impact of number of enclaves

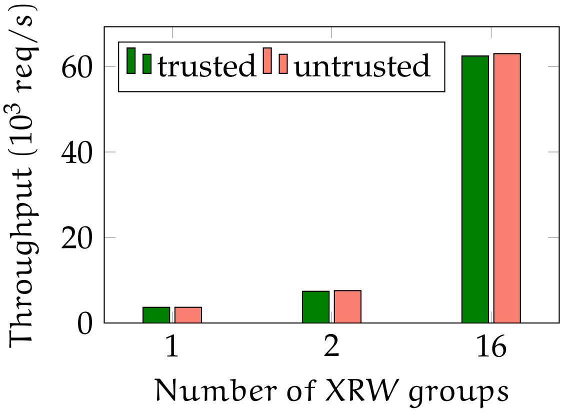


Figure 2.30: Impact of enclaving

difference between one and two enclaves. As described earlier, the `XMPP` actors need to exchange data, and in the case of the single enclave service, this data is stored inside a non-encrypted private EOS. In the case of multiple enclaves, the data is stored in an encrypted public EOS. The overhead of the encryption can be estimated at 6.2%.

The impact of the enclaving was also evaluated. By the framework's design, `XMPP` actors can be configured to work as trusted or untrusted. Untrusted `XMPP` actors are working non-enclaved and cannot use an encrypted EOS and thus, they were compared with the single-enclave trusted configurations.

During the evaluation, 6 different configurations were compared. For the constant number of clients (1000), three numbers of XRW groups (1, 2, 16) were compared for trusted and untrusted executions. Figure 2.30 shows the results. In all tests, the untrusted executions demonstrated slightly better performance compared to the trusted executions. However, this difference is insignificantly small and never exceeded 1.0%.

2.2.5 Related works

Frameworks and programming languages which feature the actor model have a long history [97, 98, 99, 100]. However, there is no related work that has proposed a framework tailored towards the use of trusted execution. For example, the CAF framework [82], which evolved from the libcppa [101] project, is an actor-based programming framework written in C++. It offers a high-performance, lightweight messaging system and supports heterogeneous actors that can interact with devices such as GPUs. However, the CAF framework misses any support for trusted execution, as well as central ideas of *E*Actors like effective enclave-to-enclave communication, flexible reconfiguration and more. Java extensions, like Kilim [102] and Akka [83], as well as languages like Erlang [100] offer an actor-based execution model based on lightweight threads and fast communication using queues. None of these frameworks supports trusted execution, and porting such heavyweight runtimes like the JVM or the Erlang VM to SGX is a challenge in itself.

As previously presented, a couple of works have already identified shortcomings of the SGX SDK: for example, HotCalls [21] offers hand-crafted spin-locks for SGX to reduce the synchronisation overhead, and SCONE [25] as well as Eleos [22] aim at avoiding costly enclave exits. However, none of these works addresses multi-enclave settings. The same applies to Glamdring [78], which enables automated partitioning of legacy applications but also does not offer support for multiple enclaves.

More related to the framework is Panoply [57], which offers lightweight enclave-based *micro-containers*. Panoply provides an environment for execution of legacy applications inside enclaves using a tiny shim layer which maps unavailable inside enclaves OS abstractions to the ECALL interface. The shim layer together with untrusted software supports the UNIX `fork` API. For this system call, Panoply creates a new enclave with the same code as the parent enclave, establishes a communication channel between these two enclaves, and performs data migration from one enclave to another. This inter-enclave communication support requires a custom mapping using a reference monitor and is as costly as using the SGX SDK. Furthermore, Panoply misses configurability, as offered by the presented *E*Actors framework.

There is a growing number of middleware-like systems which utilise SGX: VC3 enables map reduce [103], SecureStream provides tailored support for stream processing [104], and SecureVertex enables secure cloud microservices [105]. To connect enclaves, these systems use standard communication mechanisms like TCP/IP sockets (VC3), third party message queues like ZeroMQ (SecureStream) or hand-crafted event buses (SecureVertex). In essence, none of these projects offer fine-grained and fast support for multi-enclave programming, since they are designed for multi-server setups.

In line with the secure multi-party computation use case, there is a limited number of works which specifically focused on the combination of SMC and trusted execution. Iron [106] provides a practical functional encryption scheme involving several enclaves. To achieve this, intensive message exchange is required, initiated and managed by different components of the Iron platform. Communication between enclaves is performed using standard ECALLs. Accordingly, Iron could profit from *E*Actors' fast inter-enclave communication support.

Another SGX-based protocol for multi-party computation was introduced by Bahman et al. [107]. This protocol has two phases: a preparation phase and an online phase. During the preparation phase, parties, represented by SGX enclaves, establish encrypted communication channels. During the online phase, they evaluate built-in functions. Overall, their proposed SGX multi-party computation protocol and the SMC use case implemented in subsection 2.2.3.2 share some similarities. However, the work of Bahman et al. misses support for fast inter-enclave communication and system support as provided by *E*Actors.

At a higher level, the *E*Actors framework is inspired by the conception of separation of mechanisms and policies [108, 109]. Mechanisms in the framework are functions unavailable inside enclaves and thus provided by system actors. Enclaved actors, in turn, play a role of policies. This separation contributes to scalability of framework-based applications. Additionally, the framework supports the ideas of *lateral trustworthy apps* [110]. These applications are independent components of a system protected by SGX enclaves. This protection should increase security and isolation applications, which echoes with the motivation of *E*actors.

2.2.6 Summary

The *E*Actors framework enables multi-enclave programming and interaction at low cost. This is achieved by an SGX-tailored implementation of the actor model which prevents costly execution mode transitions, offers uniform communication primitives, and can be flexibly configured for different deployments. The *E*Actors framework offers a lean programming interface leading to a framework with a small trusted computing base of less than 3.3K lines of code plus some additional libraries provided by SGX SDK. Together, this paves the way for novel privacy preserving multi-party computing schemes and strengthens the security of privacy critical services such as XMPP message exchange or SMC services. The evaluations show that multi-enclave programming previously considered costly comes almost for free, and that off-the-shelf instant messaging services can be outperformed by 1.11× up to 40×.

2.3 STANlite: a database engine for secure data processing at rack-scale level

The previous section introduced a new programming model for SGX enclaves. This model is based on *actors* – lightweight entities which communicate via messages. While the lightweight nature of these entities contributes to the decrease of memory footprint, neither actor-based nor the other programming models resolve the issue of the EPC paging (section 2.1.2).

As shown in section 2.1.2.2, physical pages of all enclaves are located inside a special region of physical memory. This region has a limit and when all enclaves use more memory than the size of this cache, heavyweight paging takes place. During this process, the system software encrypts and swaps out pages from the cache and then swaps in and decrypts the previously evicted pages. This process significantly impacts the perforce of memory consuming enclaved software, particularly in-memory databases.

In-memory databases, such as Redis [111], Memcached [112], Apache Ignite [113] and others, are very popular elements of cloud systems. These databases can process security-sensitive data and thus should be protected from a cloud provider. One common approach is to use homomorphic encryption, which is slow and limited in SQL expressions [114, 115]. Under the protection of the Intel SGX architecture, enclaved databases can provide the same functionality as a non-enclaved database, but securely. However, enclaved databases should mitigate the negative impact of EPC paging.

This section addresses the issue of EPC paging in memory consuming applications such as in-memory databases. It introduces STANlite – an enclaved database for secure data processing in clouds which features in-enclave software-based paging.

The section is organised as follows. Section 2.3.1 analyses memory management of in-memory databases and provides the general view of software-based memory virtualisation. Section 2.3.2 covers architectural details of STANlite, including the virtual memory engine and the communication layer. Section 2.3.3 describes implementation details of major components. Section 2.3.4 evaluates STANlite using various benchmarks. Section 2.3.5 discusses related works, and section 2.3.6 summarises the section.

2.3.1 Towards an enclaved database with software-based paging

An in-memory Database (DB), like any other memory consuming application executing inside an enclave, faces performance degradation after reaching the EPC capacity. Software-based paging, i.e. the paging when the enclaved software performs swapping of pages without the involvement of the system software, can reduce the performance degradation. This is possible because enclaved software can avoid asynchronous exits, which are necessary for hardware-based page eviction (section 2.1.2.2). Also, in contrast with the system software, which uses the same page-eviction policy for all EPC pages, the enclaved software can use a more efficient policy tailored for a particular application. For this, however, one needs to analyse the memory usage patterns of a database, develop a Virtual Memory Engine (VME) and page eviction policies. The first task is considered by this section, while the next section describes the Virtual Memory Engine (VME), and page eviction policies.

2.3.1.1 Memory management in databases

Before the analysis of the memory usage patterns, one needs to select a database engine for enclaving. The SQLite database has been chosen for this purpose. In contrast with many in-memory databases like Redis [111] and Memcached [112] that offer a simple key-value interface, this database features rich SQL syntax, which is important for data processing in clouds. At the same time, in contrast with other relational databases like MariaDB [116] or PostgreSQL [117], this database has a very low footprint, because it was developed for embedded systems. However, since SQLite is a library database, it does not have a network layer, and it will need to be developed.

An in-memory SQLite DB uses three kinds of memory with different characteristics[23]. Firstly, there is the static memory for the DB's code and data sections. This memory is long-lived since it is allocated when the DB starts, and freed when the DB exits. Secondly, the dynamic memory is used for processing queries. Incoming queries require memory to store temporary data, and this is allocated dynamically by a DB. This memory is short-lived since it needs to be allocated and freed for each SQL request. Thirdly, there is the memory used for storing the content of an in-memory DB. This memory is dynamically allocated, but in contrast with the request's memory,

---

[23] SQLite version 3.18.2 was used for the analysis

it is long-lived. A DB frees this memory only when the content is no longer used, for example when a user drops the corresponding tables, rows, or the whole database.

During the operation, the ratio between these memory types changes. In the beginning, memory used by the DB's code and pre-allocated variables is the biggest because the DB does not process incoming messages and has no content. Later, during the active operation, the content size grows and the share of the pre-allocated variables and binaries becomes much less than the DB's content. For example, a fully functioning in-memory SQLite DB with a 200 MiB content can use less than 1 MiB of memory for code/data sections[24], and several megabytes of memory as a heap. The dynamic memory allocated for request processing varies differently in accordance with the request type. However, for most of the requests, this memory is negligible compared to the size of content memory.

These three kinds of memory have not only different sizes, but also different access patterns and connectivity with other components of a DB. For example, a DB may create linked lists inside dynamically allocated memory to process incoming requests. Movements of elements of the list may lead to its corruption. Thus, this data has high connectivity since the correctness of the whole data structure depends on the location of each element. Eviction of high-connected elements leads to the frequent swapping, while eviction of low-connected elements does not.

Memory used to store the DB's content, in turn, is much less connected with the DB engine. This memory mimics the content of a DB's file stored on a disk, and access to this data is performed via *read* and *write* operations. The elements stored on the disk are independent and do not refer to other elements directly.

Any of these kinds of memory can be software virtualised. However, the static memory has much less size than the EPC border, and thus, there is no reason to virtualise it. The heap memory used for request processing is short-lived, significant depending on the request type and DB size, and has high connectivity with the DB parts. Therefore, this memory can be virtualised, but because of high connectivity, the virtualisation may not lead to significant performance improvements. Finally, the most appropriate for virtualisation is the storage memory because elements of this memory are long-lived and independent.

---

[24] https://www.sqlite.org/footprint.html

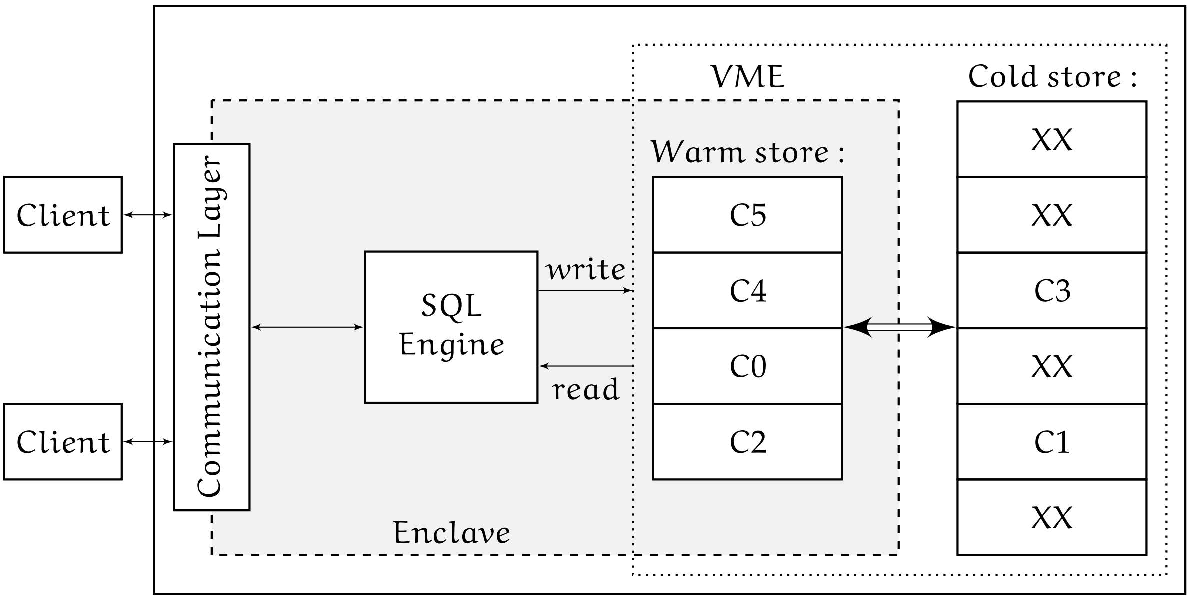


Figure 2.31: Architecture of STANlite

2.3.2 Design

Figure 2.31 depicts the general architecture of STANlite – an enclaved database with software-based paging. As can be seen, the central component of the DB is the SQL engine, which is located inside an enclave. The SQL engine uses a *Communication layer* for interaction with network clients. This layer is split between trusted and untrusted areas and supports two kinds of communication: ECALL-based TCP/IP and call-free zero-copy Remote Direct Memory Access (RDMA).

The SQL engine does not store the DB data inside its own memory but uses memory provided by the Virtual Memory Engine (VME). The VME manages memory pages which contain the DB's data and securely distributes data between in-enclave memory (so-called *warm* store) and untrusted external memory (so-called *cold* store). The VME provides different policies which impact the STANlite performance (section 2.3.3.3).

2.3.2.1 Threat model

The threat model of the STANlite DB is similar to most of the SGX-enabled systems. It is assumed that an attacker may have privileged access to software and hardware components of a cloud platform [118, 119, 24]. It is also assumed that the STANlite DB is executed on an SGX-enabled platform. Elements of this, as well as CPU instructions, operate as defined in manuals. STANlite cannot prevent exploitation of known and unknown software and hardware flaws. Side channel attacks (section 2.1.3.2), such as paging-based [60], cache-based [64, 61] or syntonisation-based [62] attacks, are also beyond the scope of this project.

2.3.2.2 Communication layer

Enclaved software is not permitted to use system calls (section 2.1.2.1). Instead, it should use ECALLs and OCALLs or system layers of frameworks like *E*Actors (section 2.2) or SCONE [25]. However, STANlite uses its own ECALL-free system layer. This is firstly because the frameworks increase the footprint of enclaved software, which is undesirable because of the paging, and secondly because these frameworks do not support a modern zero-copy network technology Remote Direct Memory Access (RDMA), which is widely used in a cloud infrastructure. This bypasses the kernel network stack by offloading networking to an RDMA-enabled device. Thus, STANlite can save processing time used for copying buffers, and increase the throughput when using RDMA for communication with clients.

Unfortunately, RDMA cannot directly write to enclave memory. However, a significant amount of time can be saved by just eliminating the additional interaction with the kernel. Furthermore, because enclaved software can directly access network packages delivered by RDMA, an untrusted helper does not need to deliver packages via ECALLs. Instead, it needs to notify the enclaved DB about incoming data, which can be done in a more effective way than delivery of a package. Subsection 2.3.3 describes this process in detail.

2.3.2.3 Virtual Memory Engine

The VME enables virtualisation of memory used by the SQL engine. In general, the VME consists of two components (Figure 2.31). The first component is a *cold* store, located outside the enclave. A cold store consists of evicted virtual pages that are stored in encrypted form. If the SQL engine requires data stored in an evicted page, the VME retrieves the page from the cold store, decrypts it inside an enclave and delivers the content of the page into the SQL engine.

The second component of the VME is a *warm* store. This consists of actively used pages but can be disabled in accordance with a VME policy. When enabled, the store is located inside an enclave in plain form, i.e. in contrast to the cold store, accesses to pages stored inside it do not require the use of encryption and decryption. The size of the warm store is defined at compilation time and can be arbitrary, but needs to be smaller than the size of the EPC minus the heap size and the size of code/data sections of the database.

2.3.3 Implementation

This section provides details regarding the implementation of STANlite. First, it starts with a description of an internal structure of SQLite, which is used as a basis for STANlite. Then it describes the glue layer between the VME and the SQL engine. After that, the section outlines VMEs and the devised communication layer.

2.3.3.1 SQLite as a core part of STANlite

The embedded database SQLite has a layered design, is configurable and self-contained. The SQLite has not been modified, although two additional layers, the *communication* layer and the VME, were added (Figure 2.32). The *communication* layer interacts with the network subsystem and passes incoming queries to (or retrieves answers from) the SQLite *Core*. The Core consists of a *SQL Compiler* and a *Virtual machine* [120]. The first component compiles incoming requests into bytecode, which can be processed by the virtual machine.

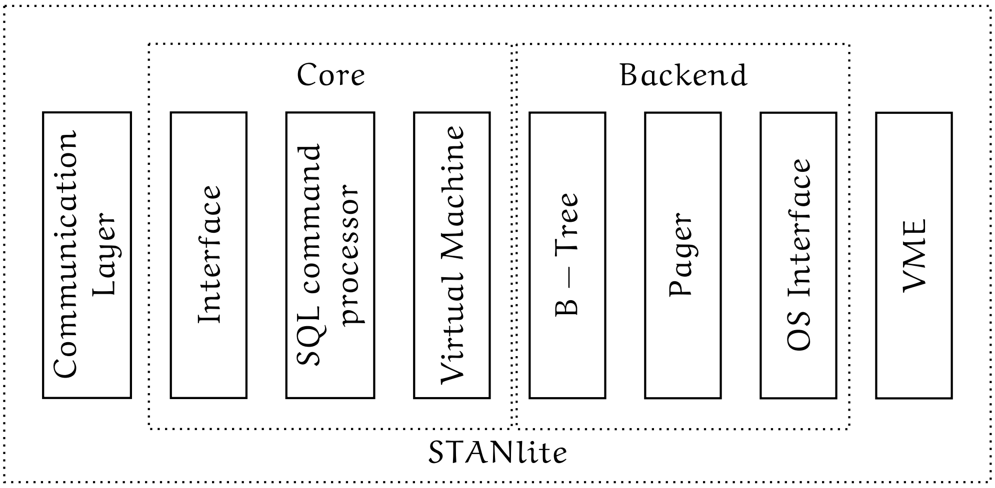


Figure 2.32: Internal components of STANlite

The Core does not store the DB's data, but instead utilises the *Backend*. The Backend implements necessary functions to interact with stored data. It uses *B-tree* structures to maintain data, and the *Pager*, which works as a page cache for actively used pages. The lowest layer is the *OS interface*, which interacts with a file-system or a storage device. The in-memory form of SQLite also uses the page cache. However, in contrast with a file-based form, SQLite never drops cached pages. In other words, the in-memory form of SQLite stores all data inside the Pager.

Two of the lowest layers of SQLite's abstractions can be used for memory virtualisation: the Pager and the OS Interface. While these layers have different functions (the Pager is for caching, the OS Interface is for interaction with files), both of them can be used to store data. Moreover, it is also important to note that the OS layer uses a Read/Write access pattern for interaction with storage.

After the in-depth analysis of SQLite, the lowest interface has been chosen for the VME implementation, since this layer is more isolated, independent and more flexible than the Pager. The VME itself was implemented as an independent module and connected to the SQL engine by glue code of the OS interface.

2.3.3.2 Glue code

The SQL engine interacts with the storage layer at segment granularity. The size of a segment is defined by the SQL engine and should be chosen in accordance with the storage sector size (or the page size in the case of an in-memory DB). During writing, the engine prepares a buffer and requests the OS layer to write the content of the buffer at a specific place on a disk. During reading, the engine prepares a read buffer and requests the OS layer to fill the buffer with data from storage at a particular offset. Data sizes in both cases are equal to the segment size.

The OS Interface used for interaction of the SQL engine with the OS layer was used as glue code to bridge the SQL engine with the VME. The glue code implemented the necessary API, while the VME implemented the storage function.

2.3.3.3 Virtual memory engine

The Virtual Memory Engine (VME) is implemented in accordance with Read/Write access patterns used by the OS Interface. The VME can be logically separated into two parts (Figure 2.33). These parts are located in different areas of a host process address space. The first part is trusted and located in an enclave. This part is active since it includes modules which actually perform memory virtualisation. The second is untrusted and located outside the enclave. This part is passive since it is used only to

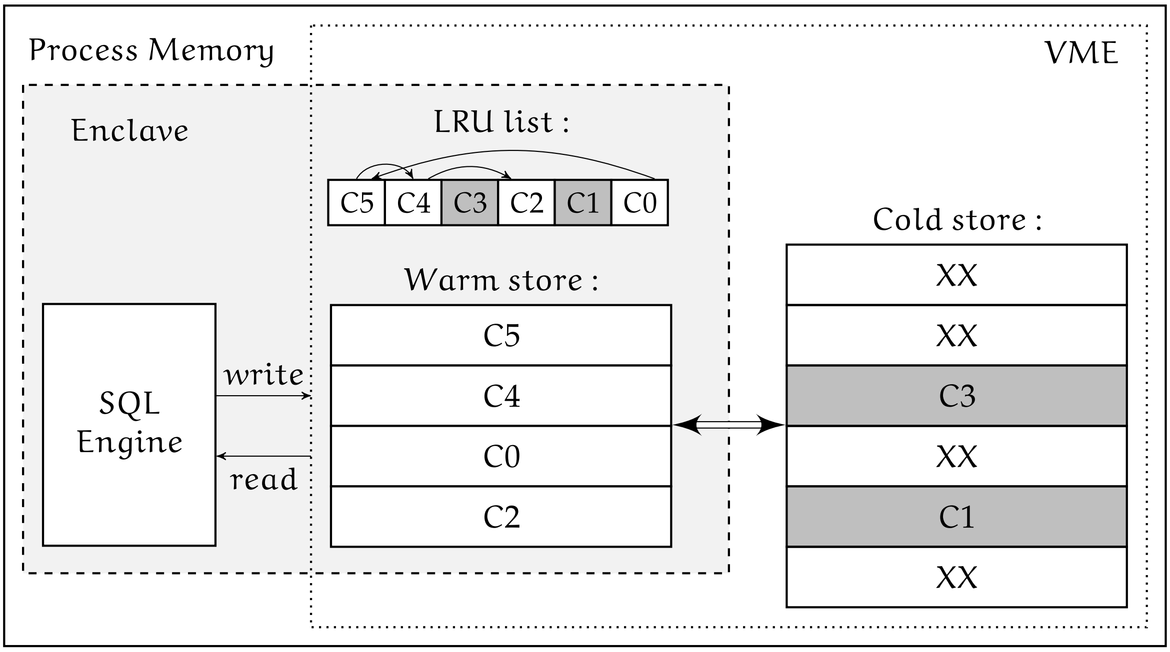


Figure 2.33: Internal structure of a Virtual Memory Engine

store the evicted pages in a memory area called *cold store*. The trusted part also can have its own memory region with pages named *warm store*.

Both warm and cold stores are split into pages of predefined size. The warm store, if used, consists of non-encrypted pages. The cold store is filled by evicted pages. Sizes of stores are defined at compilation time by the *sector size* variable. The sector size impacts the performance of the SQL engine since it defines how often the SQL engine will interact with the storage.

The total size of the cold store can be arbitrary and can be changed during the work. The warm store, in contrast, has a fixed size with an upper limit: the total size of memory used by the STANlite binary, statically allocated memory and the warm store should be less than the EPC size (Figure 2.34). In this case, accesses to memory object allocated by the VME do not cause heavyweight paging.

The trusted part consists of several components: an array of cached pages named warm store (mentioned previously), a hash table which describes the state of all pages, and a Least Recently Used (LRU) queue, which describes the current configuration of the warm store. The queue is a doubly linked list with indexes of pages currently stored inside the warm store. The head of the queue represents the most recently used page, while the tail represents the least recently used page.

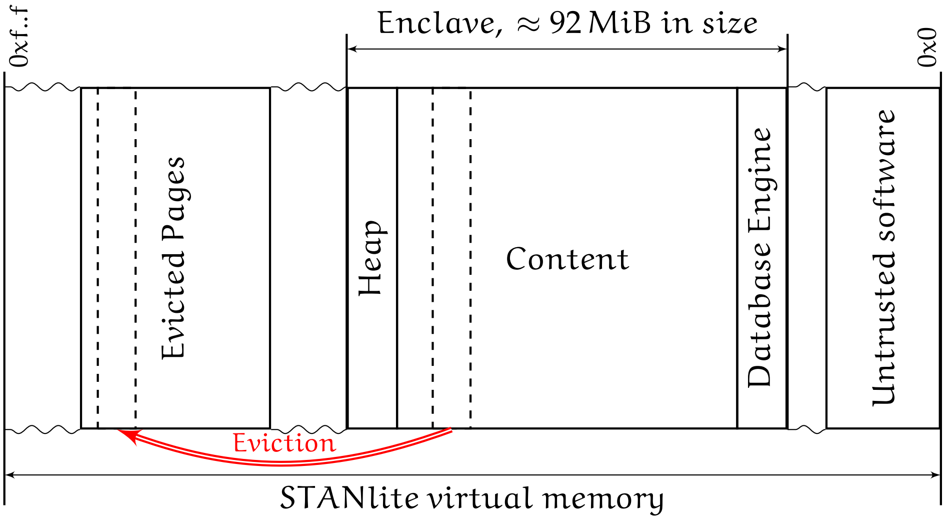


Figure 2.34: Software-based paging of STANlite

When the SQL engine writes or reads a page, the VME performs the following steps. Firstly, the VME checks inside the hash table where the requested page is stored. If the page is stored inside the warm store, then the VME performs the requested operation with the stored page and updates the LRU list. Secondly, if the requested page is located inside the cold store and there is at least one empty slot inside the warm store, then the VME performs *light paging*. For that, the engine decrypts the requested page into an empty slot of the warm store, then updates the hash table and the LRU list, and performs the requested operation. Thirdly, if the page is located inside the cold store, but the warm store is full, the VME performs *heavy paging*. For that, the engine evicts the rarely used pages from the warm store to the corresponding location of this page inside the warm store, decrypts the requested page into the empty slot of the warm store, and updates the LRU list.

To prevent *replay attacks* [121], before eviction of a page, a VME computes a SHA224 hash sum of the page's content. Computed hashes are saved inside the hash table, which describes states of all virtual pages. Later, when the previously evicted page is swapped in, the hash sum of the decrypted page will be computed and compared with the previously stored value. STANlite stops its execution and detects an intrusion when the compared values are different.

A VME can be configured in accordance with the initial design considerations. The exact configuration is based on different features included or not included into a

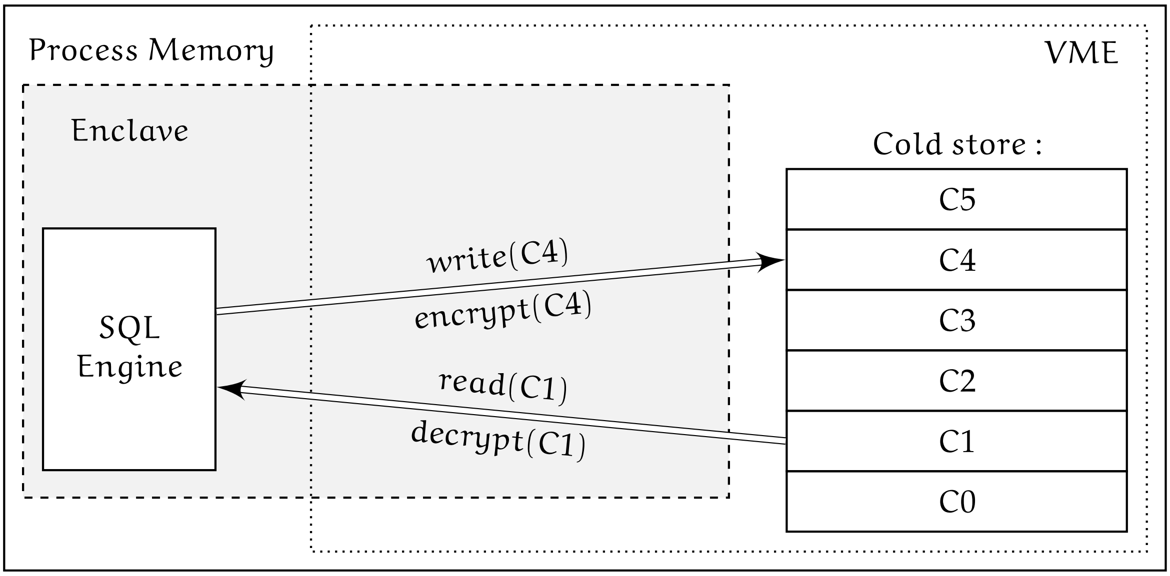


Figure 2.35: Page eviction in `--i` and `--I` VMEs

VME. A developer can configure the following features: *encryption*, *caching*, and *fetching*. When combined, it can enable one of four policies (VME modes), considered below.

Encryption

All VMEs provide integrity protection by the computing of hash sums of evicted pages. This feature cannot be disabled. However, confidentiality protection is a configurable feature: a developer can disable it. If enabled, a VME performs encryption of pages before the eviction. Otherwise, it evicts pages in a plaintext form. Respectively, on top of this feature, STANlite offers two VME modes: the first one named *integrity-only* mode, referred later as `--i`, and *integrity and confidentiality* mode, referred later as `--I`.

Figure 2.35 visualises interaction of `--i` and `--I` VMEs with the SQL engine and the cold store. As can be seen, this operation mode does not use the internal cache, and all read and write requests of the OS interface are translated directly into encryption and decryption write and read operations. In this mode, STANlite consumes the smallest amount of memory compared to other VMEs. This mode can be used in strict memory environments, or in cases where the caching of data does not increase the performance, or heap memory for processing of a heavy request needs to be allocated. Subsection 2.3.4 provides examples of these cases.

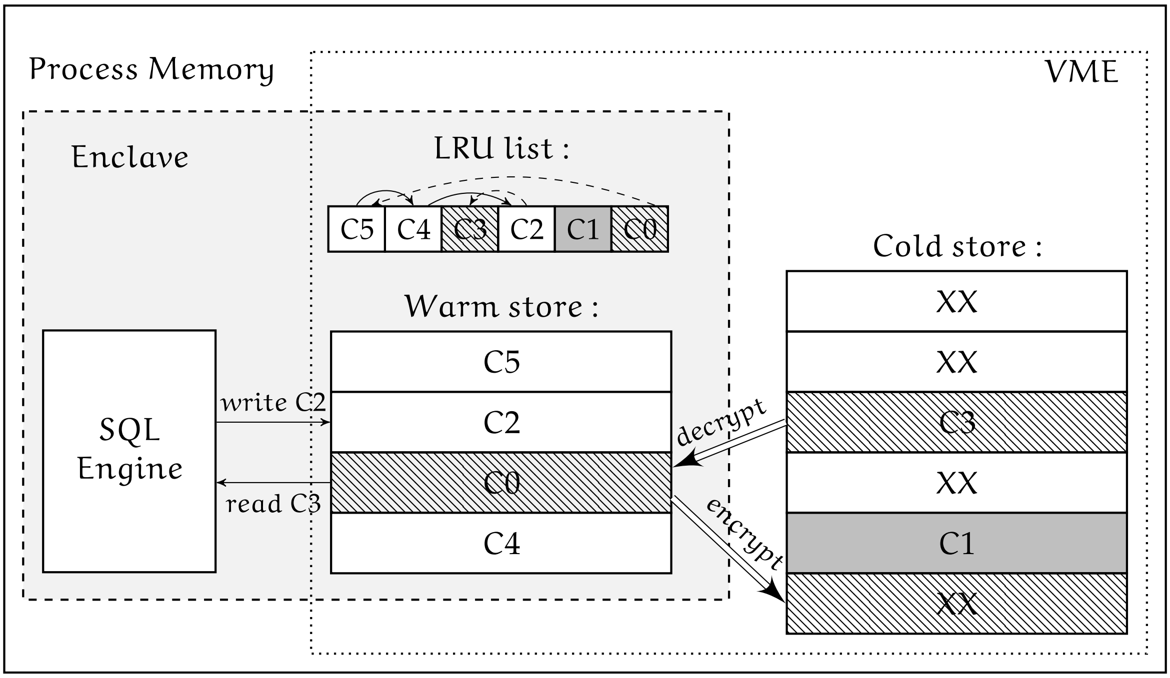


Figure 2.36: Caching and eviction of pages in the `C-I` VME

Caching

The `--I` VME can be extended by the enabled warm store. This mode is called `C-I`. Figure 2.36 visualises interaction of the `C-I` VME with the SQL engine and the cold store. As can be seen, this mode uses both the warm store and the cold store.

The figure visualises the heavy paging. As can be seen, the SQL engine first performs writing into the pre-existing warm store page (`C2`), which moves it to the top of the LRU list and then tries to read from page `C3`, which is located inside the cold store. To perform this request, the VME takes a page descriptor from the tail of the LRU list (`C0`) and then moves this page from the warm store into the cold store. The stored address is computed from the index of the page (`0`) and the size of the page (`PAGE_SIZE`). Then VME updates the record in the hash table belonging to the page `C0`. After that, the VME moves the requested page from the cold store into the empty slot of the warm store, and updates the hash table and the LRU list. After successful swapping of pages, the VME performs the requested operation with the page stored inside the warm store.

Fetching

The *Fetch* mechanism was implemented in SQLite to enable the use of memory-mapped storage devices. The Fetch call of the OS interface, in contrast to Read/Write calls, requests a memory pointer to a storage' page, but not the content

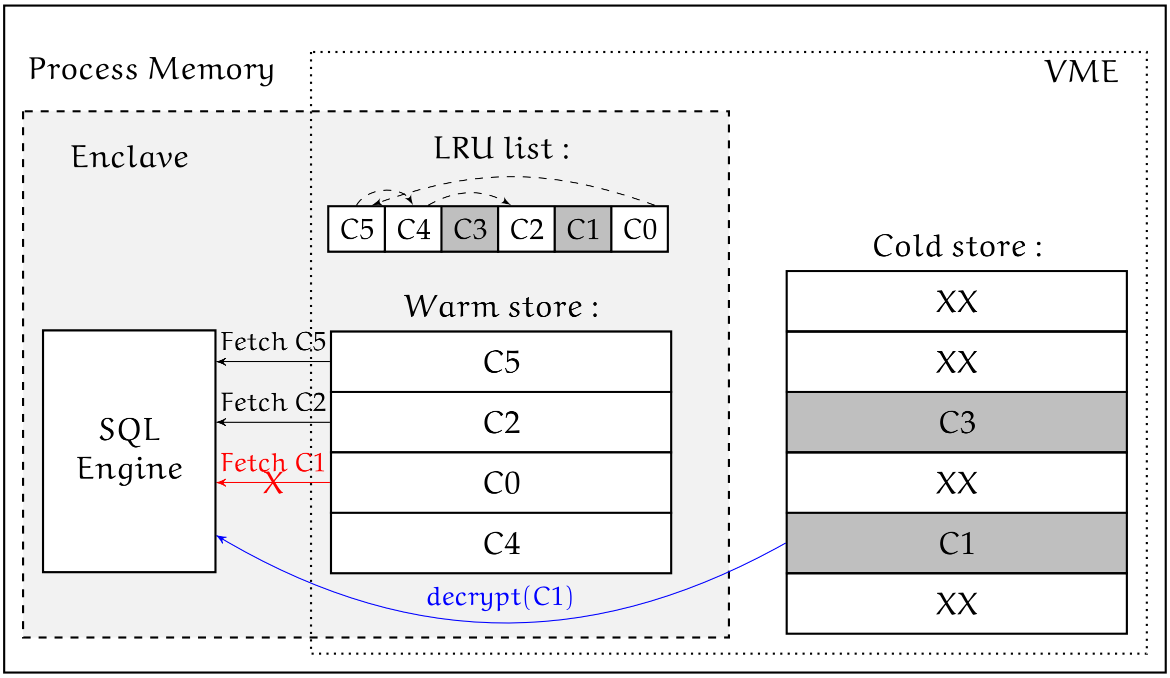


Figure 2.37: Fetching pages in the `CFI` VME

of the page. To process this request, the OS layer maps a storage page into the virtual memory of a DB and returns the virtual address. Later, the SQL engine uses this data directly, i.e. the engine skips the Page cache and some internal buffers. This design significantly decreases I/O latency.

The OS layer API uses two functions to manage *fetched* pages. The *Fetch* call requests a page, while the *Unfetch* call informs the OS layer that the fetched page can be unmapped (and additionally swapped out). If the OS layer cannot fetch a requested page, it informs the upper layer about this, and the upper layer repeats the request, but in the form of Read/Write access.

Fetch and *Unfetch* calls were implemented in the `CFI` VME, which extends the `C-I` VME. Figure 2.37 visualises this VME. As can be seen, the `CFI` mode is built on top of the `C-I` VME, i.e. it utilises the same components of `C-I`, such as warm store, cold store and the LRU list in the same way. Additionally, each element of the hash table gets a ”pinned” flag, which defines if the page can be evicted or not.

In this mode, most of the pages stored inside the warm store are ”pinned”. The LRU list cannot control accesses to these pages because the SQL engine does this directly without involvement of the OS Interface. However, the VME updates the list each time when the SQL engine issues Fetch or Read/Write calls. Moreover, after each

Unfetch call, the ”unfetched” page does not leave the warm store, but the VME puts the corresponding page descriptor into the tail of the LRU list. Thus, within the next heavy or light paging, this page will be swapped out.

Figure 2.37 shows the particular situation when all pages located inside the warm store are ”pinned”, but the SQL engine requests a page which is located inside the cold store. The pinned pages can be used by the SQL engine and thus, if they are swapped out, the database can be corrupted, because after that, the SQL engine will use data from a wrong page. However, STANlite does not stop working in this case because read and write operations can be still performed, even if the warm store is ”blocked” and cannot provide a slot for a page.

As can be seen in the figure, all pages `[C4, C0, C2, C5]` of the warm store are pinned. The {`Fetch, C5`} call can be successfully performed because the page `[C5]` is located inside the cache. However, the {`Fetch, C1`} or {`Read, C1`} requests cannot be performed because the `[C1]` page is located inside the cold store, and the warm store does not have an empty slot. In the case of this Fetch call, the VME informs the SQL engine that the Fetch request cannot be processed and after that, the SQL engine repeats the request but in the form of Read or Write access. Then, the VME transforms this request into direct encryption or decryption operation on the data stored inside the cold store. In other words, when the cache is overloaded, the `CFI` VME works in the same way as the `--I` engine until the SQL core will not release any page from the warm store via the Unfetch call.

2.3.3.4 Networking

The communication layer consists of three components. The first is responsible for data transferring at the client side (Figure 2.38). The STANlite client library is linked with the source code of a client program and implements data transfer between the program and a STANlite server. The network library supports RDMA and TCP/IP -based networking, and the data transfer has the same algorithm in both modes: the network library receives an SQL request in plain text form from the client, encrypts it and then transfers it to a STANlite server.

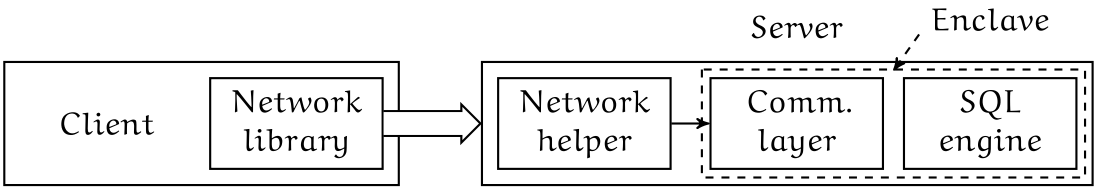


Figure 2.38: General design of networking

The second component of the network layer works on the server side. This component is a network helper which receives encrypted packages from clients and transfers these into the enclave, or receives a package from the enclave and transfers it into the network.

The third component is the STANlite network server located inside the enclave. The server receives an incoming request, decrypts it, processes it inside the SQL engine and sends a response back.

TCP/IP networking

Figure 2.39 shows the data migration path for the TCP/IP-based network layer. The data migration path includes several buffers and requires several *memcpy* operations. The initial data is presented in a plain text form and located inside a Client Plain text Request Buffer (CPRB). The client network library encrypts the content of the buffer and places the encrypted request into a Client Encrypted Request Buffer (CERB). Then the library sends the content of the CERB via TCP/IP socket. This operation requires the copying of data using the *write* system call.

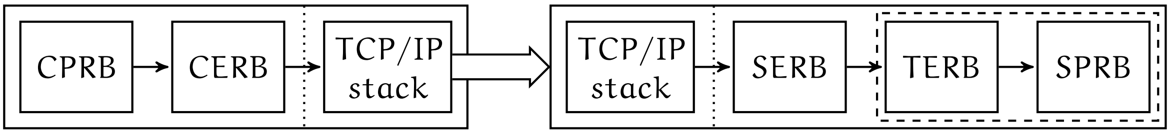


Figure 2.39: TCP/IP-based networking

The network helper on the server side reads an encrypted request from the TCP/IP socket via the *read* system call and places data into a Server Encrypted Request Buffer (SERB), which is located inside the untrusted memory. Then, the encrypted request needs to be delivered inside an enclave by the use of ECALL. The interface for communication between untrusted and trusted environments is generated by the

Intel SGX SDK. The ECALL copies data into the enclave (the internal buffer is named Trusted Encrypted Request Buffer) and then, the trusted callback decrypts the request into a Server Plain text Request Buffer (SPRB), and sends the SQL request into the SQL engine.

If the SQL engine generates an answer, the same operations need to be performed in reverse order: the response needs to be encrypted, sent via OCALL, and then delivered via TCP/IP sockets back to the client.

RDMA networking

RDMA communication does not involve the kernel stack in data transfer. Instead, RDMA-capable network devices transfer data from the virtual memory of a client to the virtual memory of a server. Figure 2.40 shows the data path for the RDMA-based communication. As can be seen, the RDMA-based communication uses the same buffers as the TCP/IP-based networking: the initial request is stored inside CPRB, and then the library encrypts it into CERB. In accordance with the RDMA specification, this memory buffer is *registered* with the RDMA device. The buffer of the same size but at the server side (SERB) is also *registered* with the network device. The data transfer between CERB and SERB is performed over RDMA by the issuing of a system call at the client side. This call does not copy data, but informs the network device that the data can be taken from the memory via DMA call and sent.

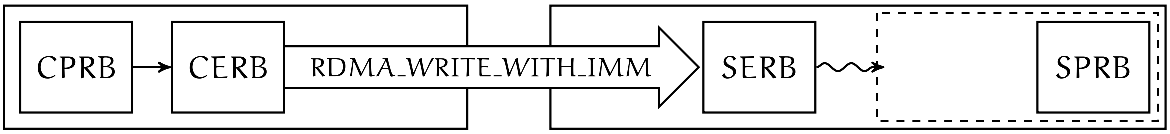


Figure 2.40: RDMA-based networking

The data delivered at the server side is accompanied by a notification[25]. The notification not only informs the server about new data but also includes the size of the data. Then, the RDMA helper does not copy the data via ECALL but uses a spinlock variable shared between trusted and untrusted areas (Figure 2.41). As mentioned previously, the enclaved code can access the untrusted memory and thus, two threads can use the same variable located inside untrusted memory as the synchronisation

---

[25] The client sends data with the RDMA_WRITE_WITH_IMM flag

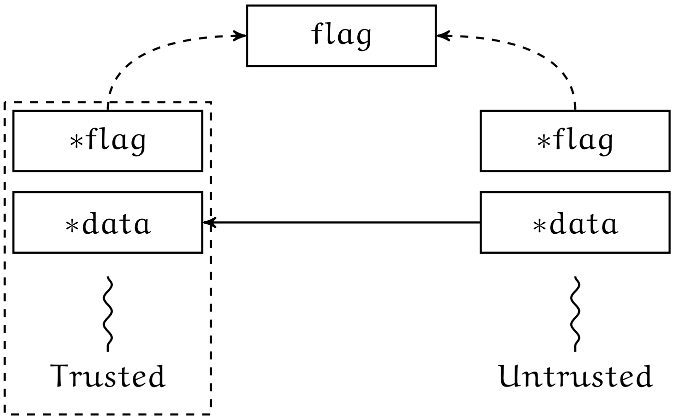


Figure 2.41: Data copy based on flag polling

primitive. The spinlock variable prevents the corruption of shared data accessed in parallel by the RDMA helper and the enclave service.

When the enclave service detects that the access to network memory is unlocked, it decrypts the package into SPRB and then processes the request. The response of the SQL engine is placed into the same buffer as the request, and the RDMA helper sends back the answer.

Integrity

For the compatibility of the RDMA and the TCP/IP network layers, the encryption of network traffic was implemented as an independent library on top of the communication layer. The library is linked with the client and trusted server source code and thus, the untrusted helpers do not have access to the requests.

The traffic encryption was implemented on top of the AES-GCM-128 primitives provided by the Intel IPP library. To send an encrypted request, a sender encrypts the plain data of the request, generates the HMAC and increments the initialisation vector IV. Encrypted data is sent together with the HMAC and verified on the server side. After decryption, the server also increments the IV of this network connection. The incremented IV protects the connection from replay attacks, while HMAC guarantees the integrity of messages.

2.3.3.5 Dynamic reconfiguration

The warm store size is defined at compilation time. As mentioned in section 2.3.1, it is assumed that STANlite mostly processes queries like `SELECT`, `INSERT` or `DELETE`, i.e. queries which do not require a lot of heap memory. However, as identified during the evaluation, there are several requests, such as `CREATE INDEX`, which require extra heap memory. The mechanism of dynamic reconfiguration was implemented in STANlite to prevent heavyweight paging of the heap memory. This mechanism can be activated by the `PRAGMA` request and works as follows: STANlite evicts all pages stored inside the warm store, releases the memory used by the warm store, and then switches the VME into `--I` mode. This mode does not use the cache, and all memory used previously by the warm store becomes available for the heap.

2.3.4 Evaluation

The following question was addressed in this evaluation: *How efficient are the VMEs compared to hardware-based paging?* To answer this, three benchmarks were developed.

The first one is named *microbenchmark*. This benchmark measures the performance of STANlite with different VMEs. During this benchmark, STANlite executes simple SQL requests and measures the response time required to process them. This benchmark shows the impact of a DB's size on the request performance. Enclaved and non-enclaved versions of SQLite databases are used as the baseline in this benchmark.

The second is named *macrobenchmark*. This is based on the *speedtest1* [122] benchmarking suite. Different SQL requests are processed differently by the SQL engine, and design of a particular VME may have an impact the performance. This benchmark evaluates the VMEs by different SQL requests, i.e. it measures the performance of STANlite for requests of different types, and compares the results with enclaved and non-enclaved versions of SQLite. The benchmark uses all implemented VME modes (including integrity preserving `--i`), different payload sizes of requests, and different sizes (smaller and bigger than the EPC size) of testing DBs.

The third one is a complex TPC-C [123] benchmark which implements an abstract billing system of an industry service with multiple users. This benchmark evaluates the VMEs by realistic sequences of SQL requests. Moreover, the benchmark requires communication layers of both kinds and uses different VMEs.

2.3.4.1 Platform and configurations

The following hardware was used in the evaluation. Two identical servers based on Intel Xeon CPU E3-1230 (3.40GHz, 4 cores, 8 hyper-threads), equipped with 32 GiB of RAM and Mellanox MT27520 RoCE RDMA controller (10 Gib). The same network cards were used for TCP/IP and the RDMA-based types of communication. Ubuntu GNU/Linux version 16.04.3 with Linux kernel version 4.4.0 was used as the operating system. RDMA capabilities were supported by libraries from Mellanox OpenFabrics Enterprise Distribution version 4.1-1.0.2.0. The Intel SGX SDK version 1.8 was used as the basis for enclaved software development. Encryption primitives used for traffic and page encryptions are based on the Intel IPP library included in Intel SGX SDK. All parts of STANlite, including client code, were compiled with "-O2" optimisation flag.

Trusted Computing Base

The TCB size of STANlite is extremely low. As with any other enclaved software, the STANlite TCB includes only software located inside the enclave. This software includes only necessary libraries provided by Intel SGX SDK, call generation subsystem and basic functions of memory allocation. It does not have external dependencies. The TCB also includes encryption primitives from the Intel Integrated Performance Primitives (IPP) library[26], provided by the SDK.

The biggest component of STANlite is the SQL engine, which is based on SQLite version 3.18.2. The SQLite has roughly 110000 Source Lines of Code (SLOC). Other trusted components, such as the VME implementation, glue code between the VME, and the SQL engine, benchmarks and enclaved parts of the communication layer do not exceed 5000 SLOC together.

Memory layouts configurations

A memory layout configuration is the configuration of different memory-related objects, such as a heap size, sizes of cold store and warm store, etc. Different benchmarks have different memory layout configurations. Warm stores of all cached VMEs (`C-I`, `CFI`) is 80 MiB in size for the TPC-C benchmark, and 70 MiB for the *speedtest1* benchmark. Since the non-cached VMEs (`--I`, `--i`) do not have warm stores, this size is 0. The size

---

[26] https://software.intel.com/en-us/intel-ipp

of memory used by the STANlite binary inside an enclave is approximately 1 MiB. The baseline presented by the enclaved and non-enclaved SQLite databases has near the same size of the binary.

Both enclaved (`VNL`) and non-enclave (`NTV`) versions of SQLite are configured to work in ″in-memory″ mode[27]. The heap size available to these configurations is limited by 2 GiB. The heap sizes of all VME-based configurations are limited by 300 MiB. However, only 8 MiB of them are used in the TPC-C benchmark, and the 16 MiB heap is enough for most of the components of the *speedtest1* benchmark. Exceptions are considered in the following subsections.

Both STANlite and SQLite databases use the same segment and sector sizes – 4096 bytes. This value is a default value of the segment for SQLite, and it was chosen as optimal after multiple experiments.

2.3.4.2 Microbenchmark

The microbenchmark is based on a simple SQL `SELECT` request. The goal of this test is to compare the time of random read accesses for different implementations of VMEs and different sizes of DBs. The following configurations were compared: STANlite with three engines (`CFI`, `C-I`, `--I`), enclaved SQLite (`VNL`), and non-enclaved (native) SQLite (`NTV`). The load generator was built-in for all experiments, i.e., there were no ECALLs issued in the enclaved tests.

For this benchmark, a table with a primary key and a text field was created[28]. Then, this table was filled[29] by multiple queries with random text inside. The size of the text of each request was close to 1 KiB, and the content of 4 requests fit exactly into a 4 KiB memory page. Then, the time necessary to perform 10 random selects[30] was measured. After that, the DB was dropped, and the test was repeated with the bigger number of inserts. In total, starting from 10000 inserts, with 10000 inserts per step, 51 different tests were performed. The corresponding size of the testing DBs changed from 1 MiB up to 512 MiB.

Figure 2.42 presents the results of the benchmark. Firstly, as can be seen, there is a size of DB, after reaching of which, most of the tested configurations change their

---

[27] SQLite uses in-memory mode when the *:memory:* word is specified as the database name

[28] `CREATE TABLE STEST("ID INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, BODY CHAR);`

[29] `INSERT INTO STEST (BODY) VALUES('<..>')`

[30] `SELECT * FROM STEST ORDER BY RANDOM() LIMIT 1`

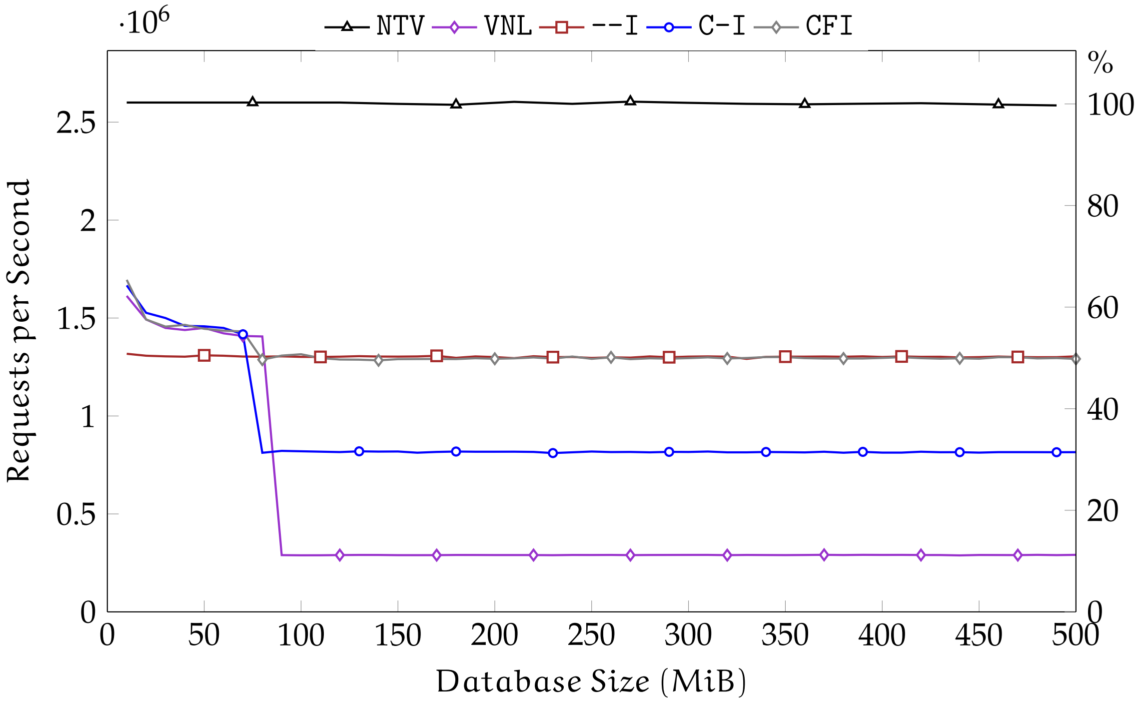


Figure 2.42: Comparison of performance in the microbenchmark

behaviour. This point is close to the EPC border in the case of `VNL`, and the border of the warm store in the case of cached VMEs. After that point, as expected, the performance of `VNL` drops significantly: before the point, the `VNL` performance is just 1.8 times smaller than `NTV` performance, but after the EPC border, the performance falls 4.5 times more and reaches 8.9 times in total. The performance of `C-I` also drops. However, this is not as significant as `VNL`. After the reaching of the warm store border, the performance of `C-I` reaches 31.5% of the `NTV` performance. The difference between `VNL` and `C-I` is 2.8 times for big DBs. This difference roughly estimates the difference between software and hardware-software implementations of paging.

Secondly, as can be seen, the behaviour of `--I` differs from the behaviour of other testing setups. Because `--I` does not have the cache, its performance is constant and does not depend on the DB size. For small DBs, the performance of `--I` is smaller than the performance of cached engines because data encryption and decryption require more time than the copying of memory from the cache into the SQL engine. However, this difference is not significant. For a database 60 MiB in size, the difference between `C-I` and `--I` is just 11%. This roughly estimates the difference between memory copying and hardware accelerated encryption/decryption. For big databases, the `--I` engine demonstrated 4.44 times more performance than `VNL`, and 1.6 times more than `C-I`. The following example analyses the sources of the performance difference for `--I` and `C-I` engines.

The `C-I` engine always uses the warm store. If the used data fits into the cache, the VME stores this data inside the enclave's memory and does not use encryption/decryption. The microbenchmark uses linear access to the stored data, and thus, after reaching the warm store border, the cache stops working because the stored data is not re-used. However, the `C-I` VME continues to use the warm store and as a result, unnecessary operations are performed. For example, for a read request, the `--I` VME performs one single operation (decrypt), while the `C-I` VME performs a combination of operations: encrypt a cached page, decrypt one page from the cold store to the same place, and then perform copying. As a consequence, for this type of requests, `C-I` shows the better performance for small DBs (compared to `--I`), and the worse performance for big DBs.

Thirdly, as can be seen, `CFI` shows the best performance for both types of memory sizes: for small DBs `CFI` behaves as `VNL`, but after reaching the warm store size, `CFI` works as `--I`. The reason for this behaviour is hidden inside the mechanism of fetching. When the SQL engine processes requests, it fetches and ”pins” them inside the cache. The engine assumes that the data will be reused later and then, fetched pages should not be evicted. When the number of fetched pages reaches the warm store size, `CFI` works as `VNL`, until an *Unfetch* call will not be issued by the SQL engine. The same behaviour can be seen in the figure.

2.3.4.3 Speedtest1 benchmark

The *speedtest1* benchmark is one of the tests included in the SQLite project. This benchmark includes most of the requests which SQLite is capable of processing. The benchmark creates several tables with different configurations and then sequentially performs various queries with them, starting from a simple `SELECT` request and ending with complex ”subquery in a result set” and four-way `JOIN` requests. Since the SQL engine of STANlite is SQLite, the *speedtest1* benchmark can be used for comparison of different VMEs.

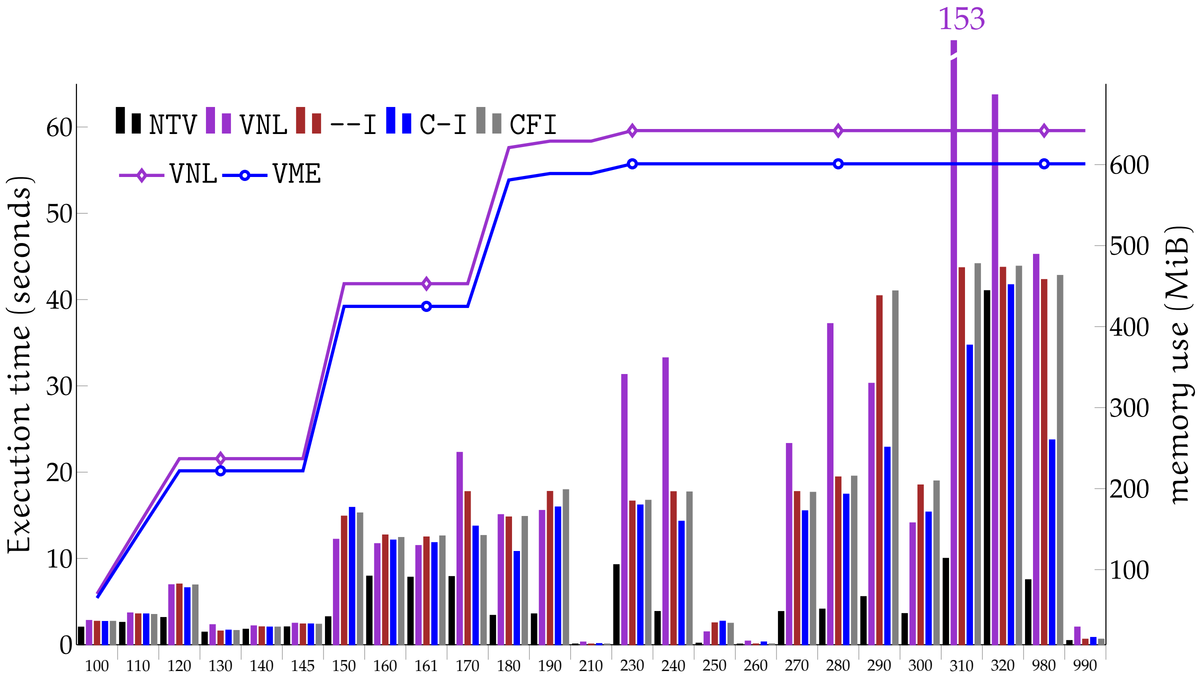


Figure 2.43: Performance of different VMEs, local execution of speedtest1

As previously, two baseline versions of SQLite were also compared: non-modified `NTV` and enclaved `VNL`. The load generation *speedtest1* program was compiled together with SQLite and STANlite. The sizes of segments and sectors were the same as in the microbenchmark – 4096 bytes. The `speedtest1 size` variable had a value of 2000. This affects the sizes of tables and the number of queries performed during tests. The payload of all requests was default, i.e. randomly generated.

Figure 2.43 shows the results of the *speedtest1* benchmark. The diagram includes the execution time of each benchmark's test for each evaluating service and a database size of the corresponding tests. Firstly, as can be seen, all STANlite engines consume less memory than SQLite. Because of the effective integration of the virtual memory engine with the SQL engine, STANlite consumes less memory. The difference is constant for all VMEs and can be estimated at 6.8%.

Secondly, as expected, there are many tests where STANlite outperforms enclaved SQLite. In some, this difference can be estimated at 1.5–2.0 times, for example in 230 (indexed `UPDATE`s), 240 (`UPDATE`s of individual rows), 280 (`DELETE`s of individual rows), 320 (subquery in a result set), 990 (`ANALYZE`). Moreover, there is the test 310 (four-ways `JOIN`s), where the difference exceeds 3.5 times. These benchmarks are characterised by the intensive memory use caused by active read/write accesses. In these kinds of accesses, VMEs work very effectively.

Thirdly, there are several tests (100-145, 160-161 and others) where all STANlite shows slightly better performance than enclaved SQLite. In these tests, the difference can be estimated at 10%–30%. Moreover, there are several tests where `VNL` significantly outperforms all (or some) of VMEs. These are: 150 (creation of indexes for tables), 290 and 300 (refills with different conditions). They are characterised by the intensive use of the heap memory to process incoming queries.

As mentioned previously, the heap memory was limited by 16 MiB for the most of the tests. However, these 3 particular tests required up to 300 MiB of heap memory. Since STANlite does not virtualise the heap memory by design, these requests expectably show degradation of the performance, since they cause heavyweight hardware/software system paging.

Finally, as can be seen, there is a difference in the behaviour of VMEs. As in the microbenchmark, the `--I` and the `CFI` engines show the same results very often, while the `C-I` engine usually demonstrates better results compared to `--I` and `CFI`. Deep analyses showed that in 290 (refills) and 980 (integrity check), the warm store significantly improves the DB performance, while in the case of `CFI`, this cache is filled by *fetched* pages.

The *speedtest1* benchmark shows that different types of requests (and their sequences) can demonstrate different performance results. For the *speedtest1* benchmark with 2000 as the size value, the best performance results were demonstrated by the `C-I` VME. The total execution time of this VME was 1.79 times smaller than the total execution time of the enclaved SQLite.

Small database

As shown previously by the microbenchmark, the VMEs behave differently for different DB sizes. The DB generated during the *speedtest1* benchmark was bigger than the EPC size. After the execution of the first test (100), which creates the first table, the size of DB was more than 100 MiB, which is definitely more than the EPC size. The same benchmark was also repeated for a smaller size of requests, i.e., for the smaller DBs.

Table 2.1: Speedtest1 total execution time, seconds

DB size	`NTV`	`VNL`	`--I`	`C-I`	`CFI`
51 MiB	6.9	8.6	24.1	13.3	13.4
601 MiB	136.5	545	373.3	305.2	370.6

Table 2.1 shows the result of the *speedtest1* benchmark performed for big and small databases. The maximum memory consumed by the small DB was 51 MiB, which is less than the size of the warm store and the EPC. The big DB, as mentioned previously, is bigger than the EPC size after the end of the first test.

As can be seen, as in the microbenchmark, the cached VMEs show better performance than `--I`. However, the results of STANlite in these tests are much worse than `VNL`, and the difference between `VNL` and `--I` exceeded 11%. These results were expected since the goal of the project was a development of a database optimised for heavy memory use.

Integrity preserving data management

VMEs can evict non-encrypted pages. The impact of encryption was evaluated by the comparison of `C-I` and `C-i` VMEs. Both engines were tested by the *speedtest1* benchmark. The configuration of the *speedtest1* was identical to the configuration used in the subsection 2.3.4.3.

As expected, the performance of the preserving integrity-only VME was significantly bigger than the performance of the `C-I` VME. The benchmark's execution time for the `C-I` engine was 1.23 times more than the execution time for the `C-i` engine.

2.3.4.4 TPC-C benchmark

The specification [123] of the TPC benchmark C (TPC-C) describes only an abstract structure and business logic of the benchmark. The queries themself, as well as a DB schema, are not included in the specification because different databases have different APIs and query types. As a consequence, there are many different implementations of the TPC-C benchmark for different databases. For reproducibility of results, the TPC-C

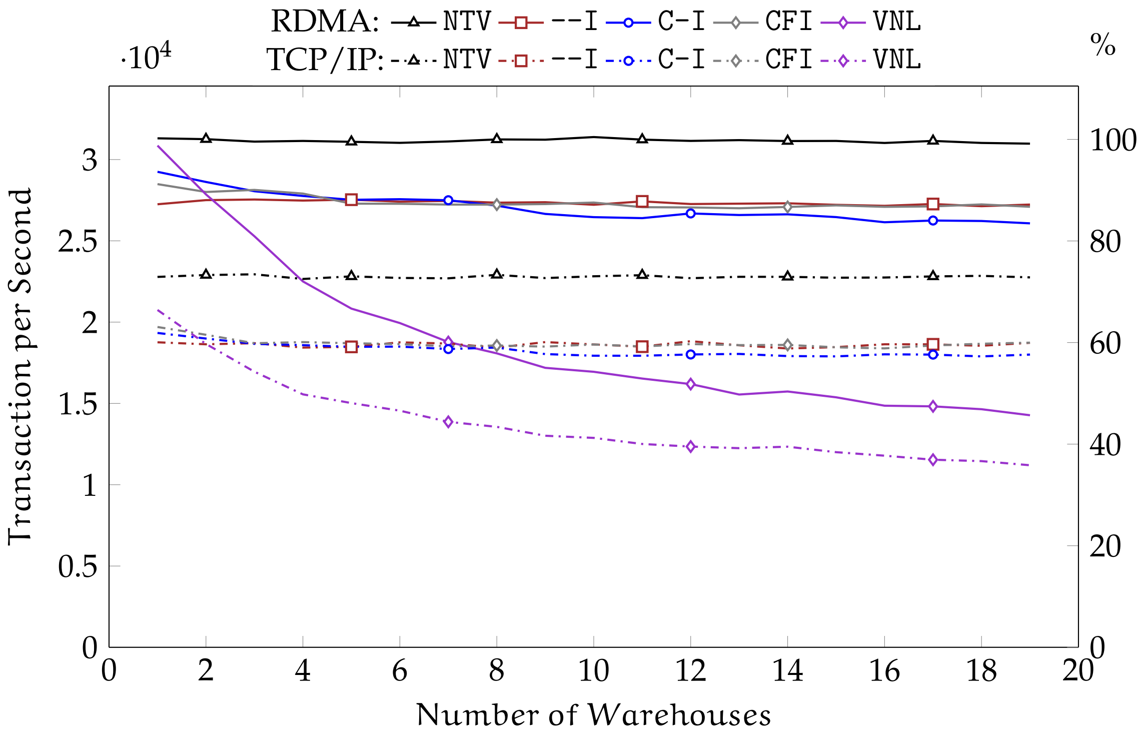


Figure 2.44: Comparison of performance in TPC-C

benchmark was not implemented from scratch. Instead, the STANlite benchmark used queries generated by the existing implementation of the TPC-C for Python[31].

The queries were generated for the different number of *warehouses* – an important characteristic of the benchmark which impacts the database size. Each warehouse increases the DB size by approximately 110 MiB. In sum, 19 experiments were generated: each experiment for each number of warehouses in the range of [1:19]. Then, these requests were processed by a testing system, and a number of Transactions per Second (TpS) was used as the performance metric.

Figure 2.44 shows results of the benchmark. It includes two baseline SQLite DBs (`VNL`, `NTV`) and three VMEs (`--I`,`C-I`, `CFI`). Two communication layers were also used: TCP/IP and RDMA.

As can be seen, the general pattern of the TPC-C benchmarks repeats the previous experiments. The performance of enclaved SQLite (`VNL`) degrades over the number of warehouses, but outperforms all other configurations at the beginning. The `--I` engine, as earlier, shows the constant performance for any size of the DB. The `CFI` engine repeats the behaviour of the `--I` engine for big databases, and slightly outperforms

---

[31] `https://github.com/apavlo/py-tpcc`

the `--I` engine at the beginning. The `C-I` mode demonstrates the best performance of Virtual Memory Engine (VME) at the beginning because of the effective cache, but later, its performance degrades, until the DB size does not reach approximately 700 MiB in size, when the performance impact of the warm store becomes negative.

Additionally, the diagram clearly shows the positive impact of the RDMA. For small databases zero-copy RDMA-based networking demonstrates near the same performance of enclaved and non-enclaved SQLite databases. Enclaved STANlite with the RDMA communication layer always outperforms the non-enclaved SQLite with the TCP/IP communication layer. In sum, STANlite with the `--I` engine and RDMA networking slower non-enclaved `NTV` at just 14.8%, while the difference between TCP/IP-based `VNL` and RDMA-based `CFI` is 2.12 times for 1 GiB DB in size, and 2.42 times for 2 GiB DB in size.

2.3.5 Related works

Pre-SGX trusted execution: MrCrypt [115] uses homomorphic encryption and processes queries in an encrypted form. CryptDB [114] also provides query-based homomorphic encryption and operates as a proxy that encrypts sensitive information at the request level. Working on encrypted data either reduces query expressiveness or substantially impacts performance. TrustedDB [124] demonstrated that a dedicated secure co-processorthat processes requests securely and independently from an untrusted host platform can overcome these issues. However, the secure co-processor used by TrustedDB enabled execution of only a lightweight database, which resulted in the hybrid architecture based on two databases: a lightweight trusted one and a full-fledged untrusted one. Cipherbase [125] also used a secure co-processor, but instead of a trusted database, it used the co-processor to simulate homomorphic encryption on top of non-homomorphic encryption schemes. STANlite has similar goals to these projects, but only relies on commodity hardware and secures processed data and code.

RDMA-based services: Pilaf [126], MICA [127] and HERD [128] are key-value stores which utilise RDMA. While these projects provide high-performance storage, they do not have any mechanisms for data protection, either during data transfer or while performing data processing. Contrarily, STANlite combines the use of RDMA with trusted execution.

SGX-based projects: Orenbach et al. introduced Eleos – the software-based paging technique for programs written in C++. A key abstraction of their design is *spointer* – a specific instance of a smart pointer which can determine if referenced data is inside or outside the EPC. STANlite shares the general direction with Eleos since both projects are aimed at software-based paging. However, STANlite focuses on custom paging support for a complex in-memory database and enables fast remote interaction using RDMA in combination with the use of SGX.

Several related projects offer execution of databases inside SGX enclaves. Microsoft SQL Server 2014 was enclaved by Haven [26] – a framework which enables shielding execution of legacy applications. EnclaveDB[32] [129], is also an enclaved database, but with a significant lower TCB, since the compatibility layer of the DB is small. In contrast with STANlite, none of these projects offer virtual memory support nor fast remote interconnection.

Panoply [57], Graphene-SGX [27] and SCONE [25] provide a general purpose trusted execution environment for legacy programs. For example, SCONE has been used for execution of enclaved Memcached. These frameworks can host an in-memory database such as STANlite, but none of them offers scalable, enclave-based paging support.

2.3.6 Summary

Enclaved memory-heavy applications, such as in-memory databases, face the problem of EPC paging when consuming more than approximately 92 MiB of memory. During this process, the system software performs page swapping, which leads to significant performance degradation. STANlite, presented in this section, demonstrated that an in-enclave software-based Virtual Memory Engine (VME), integrated into a DB, can resolve this issue since it performs page swapping without the involvement of the system software. It uses a custom VME and the ECALL-free RDMA-based communication layer. Working together, these components avoid heavyweight transitions and enable processing of large volumes of data with comparatively minimal overhead.

---

[32] This paper was published later than the STANlite project

3. Non-volatile memory and persistent systems

The memory hierarchy is a core concept of computer architecture. It arranges all storage devices in the architecture by the access time and capacity [130]. These storage devices are based on different technologies and, as a consequence, have different characteristics and features.

The top layer of this hierarchy is represented by CPU caches. The caches are the fastest memory in a system, but small (tens of megabytes) and very expensive. Moreover, CPU caches are *content-addressable* memory, and thus, cannot be addressed directly [131]. The second layer of the memory hierarchy is the main memory, which is represented currently by Dynamic RAM (DRAM). This type of memory is *byte-addressable*, volatile and high-performance. DRAM is much cheaper than CPU caches and much more capacious [132]. Modern servers can be equipped with tens of gigabytes of memory. Finally, the lowest layer of the hierarchy is secondary storage, usually represented by Hard Disk Drive (HDD) or Solid-State Drive (SSD)/flashes. This type of memory is much slower than DRAM, but significantly cheaper and capacious: secondary storage devices may store terabytes of data. They are *block-addressable* and persistent, i.e. do not require an external source of power to retain stored data.

Operating Systems (OSes) and programs rely on this hierarchy. Programs are passive entities stored inside non-volatile storage devices. However, they can be executed after they have been located into the volatile main memory. The life cycle of a program includes necessary steps to allocate memory for executables, prepare an execution context, deploy binaries in memory, and more. Executing programs produce volatile objects which can become persistent by saving into secondary storage.

The life cycles of hardware and software are also developed in accordance with the volatility of the main memory. For example, after a power reset, all devices, including

CPUs, start from default states. Then, the primary bootloader reads components of an OS from secondary storage into memory. Because of the volatility, an operating system can initialise devices and start programs, even if the previous execution has been accidentally ended by a crash or a power failure.

Technologies of non-volatile memory can change the memory hierarchy and the whole computer architecture. For example, with persistent primary storage, there is no need for data migration between secondary and primary storage devices. As a side effect, the system cannot be easily restarted in the same way as a volatile system, since programs are persistent and they need to be transformed to a some initial state [31].

Currently, there is no "de-facto" standard for the architecture of persistent software systems. Moreover, there is no single technology of persistent memory. Instead, there is a group of different candidate technologies, each of which has its own characteristics like read/write latencies and endurance. These characteristics may vary from characteristics of DRAM, thereby affecting system architecture. As a consequence, there are several competing *models of persistence*, i.e. conceptions which components of a system need to be persistent and how they need to be integrated in a system, based on different assumptions and Non-Volatile RAM (NV-RAM) technologies.

This chapter is devoted to the NV-RAM technologies and architectures of persistent systems. Section 3.1 provides an overview of candidate technologies of persistent memory and existing architectures of persistent systems. Section 3.2 describes a conception of *hypervisor-based persistence* and NV-Hypervisor – a new model of persistence and a persistent hypervisor for legacy and proprietary software.

3.1 Background

3.1.1 Candidate technologies of persistent memory

There are several candidate technologies of persistent memory. They are based on different physical phenomenons and have different characteristics. However, despite the differences, they have two common features: they can be used as a basis for *byte-addressable* memory modules, i.e. modules which support accessing to individual bytes, and *persistence*, i.e. they can retain stored data without an external source of power. In other words, these technologies can be used to create main memory modules which do not require constant powering.

This subsection provides an overview of the most prospective memory technologies such as Battery-backed RAM (BBRAM), Phase-Change RAM (PC-RAM) [133], Ferroelectric RAM (FeRAM) [134] and Magnetoresistive RAM (MRAM) [135].

3.1.1.1 Battery-backed RAM

Technologies of Battery-backed RAM (BBRAM) have been developing since the middle of the 1990s. A typical BBRAM module includes two components: an ordinary memory module and an independent power supply. A simple battery or a complex Uninterruptible Power Supply (UPS) can be used as the power supply. In case of a power failure, these batteries power the memory module for a short period of time, during which the content of the memory should be mirrored to persistent storage or main power should be recovered. Early persistent platforms like Apple Newton [39] and Rio [136] used BBRAM in their bases (section 3.1.2.2).

In the recent decades, this approach has been evolved and applied to main memory. Market available Non-Volatile Dual In-line Memory Module (NVDIMM), provided by AgigA Tech [137] and Viking Technology [138], use supercapacitors as a source of temporary power, and have new components: NAND flash and a Power Outage Detector (POD) [139]. The POD can detect a power outage in the early stages and, by the use of residual energy of a platform and the supercapacitor, can mirror the DRAM data to the NAND flash (Figure 3.1). Later, after the recovery of power, the system can recover the previous memory state from the flash.

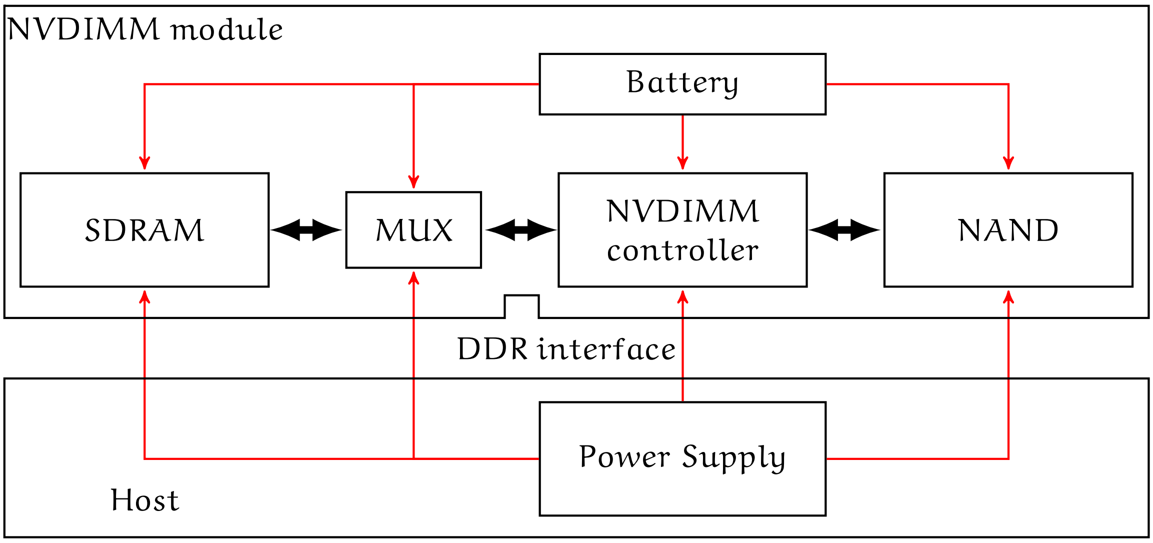


Figure 3.1: block diagram of Non-Volatile Dual In-line Memory Module

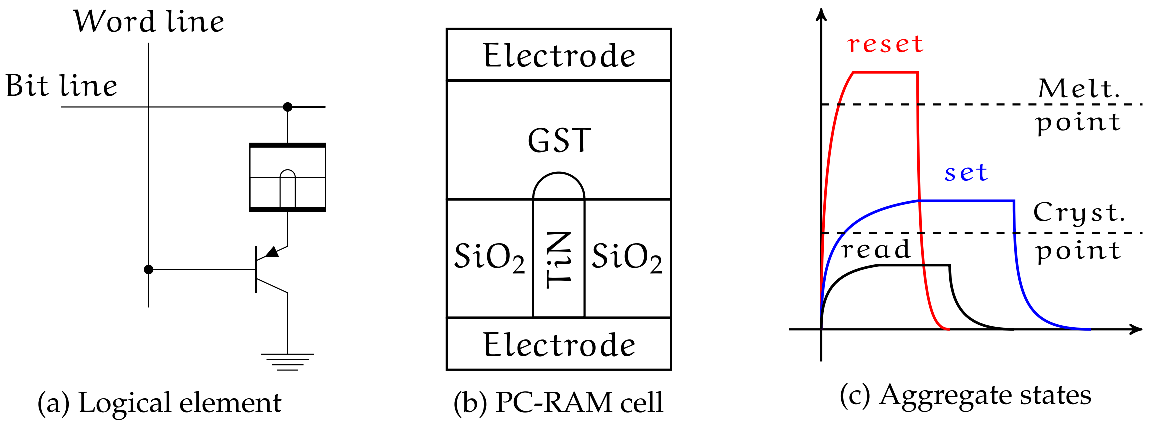
In contrast to other candidate technologies of persistent memory, BBRAM have the same read/write latencies and capacity as ordinary DRAM, since the BBRAM is based on DRAM modules. This makes BBRAM ideal for prototyping of NV-RAM-based platforms (section 3.2.1.1).

3.1.1.2 Phase-Change RAM

Phase-change memory is memory technology based on the physical properties of *chalcogenides*. The chalcogenides are a family of chemical compounds based on metals and *chalcogens* – chemical elements in the 16th group of the periodic table, like selenium and tellurium. Some chalcogenides have a unique feature: their different aggregate states are characterised by different resistances. For example, a chalcogenide Ge2Sb2Te5 (GST) has two aggregate states: a low-resistance crystalline state and a high-resistance amorphous state. This difference in resistances is used to encode a logical `1` and `0`.

Figure 3.2a depicts the structure of a PC-RAM cell. As can be seen, the cell includes a silicon insulator SiO2, a chalcogenide GST, and a heating element TiN [140]. These components are located between two electrodes attached to a transistor. Combined together, the cell and the transistor work together as a 1R-1T logical element.

The switching between aggregate states and, as a consequence, between high and low resistive states, is performed by the heating. Figure 3.2b visualises this process. The heating above the melting point destroys the crystalline structure of a GST, which then enters into the amorphous state after cooling (red line). To switch it into a crystalline



(a) Logical element (b) PC-RAM cell (c) Aggregate states

Figure 3.2: Phase-Change RAM

form, the amorphous GST needs to be heated to a temperature which is in between the crystallisation and the melting points (blue line). At this point, the GST enters into the crystalline form, which is preserved after cooling.

Switching between states requires significant time, much more than the changing charges in DRAM cells. Moreover, because of the regular heating and cooling, components of PC-RAM cells degrade over time. In consequence, the endurance of PC-RAM is lower than the endurance of DRAM [33].

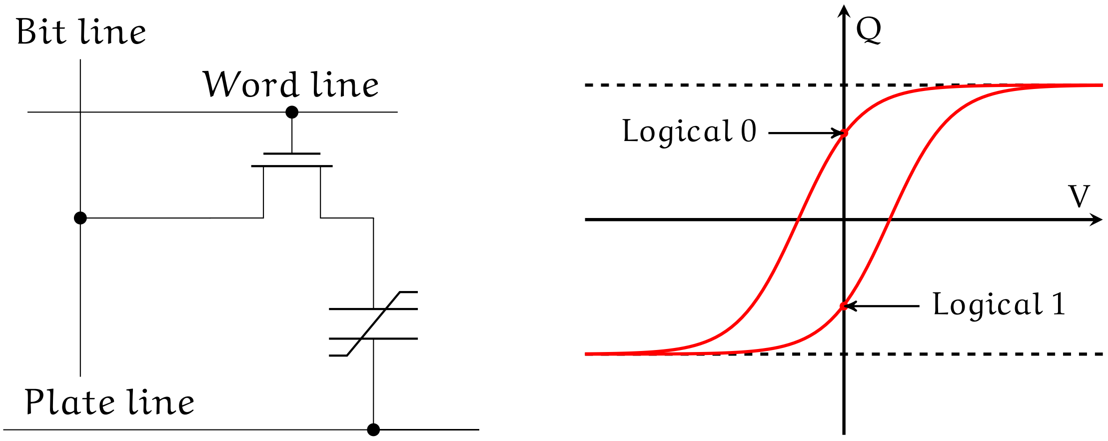
3.1.1.3 Ferroelectric RAM

Ferroelectric memory [134] is memory technology based on *ferroelectricity* – a property of some materials to have a spontaneous electric polarisation which can be reoriented by the applying of an external electric field [141]. When an electric field is applied, atoms inside a ferroelectric material shift to certain positions inside the crystal structure. These shifts change the electric polarisation of the material. Without the external electric field, these atoms keep the selected positions, and the material preserves its own polarity. Logical `0` and `1` are encoded by opposite orientations of the polarisation vector, which is defined by the polarity of charges and applied electric fields (Figure 3.3b).

A FeRAM cell has the similar 1T-1C structure as a DRAM cell (Figure 3.3a). However, instead of the dielectric used in DRAM conductors, the FeRAM cells use the ferroelectric material. During the write operation, atoms inside the material become organised by the polarity of the applied electric field. To read, a special regulating transistor tries to detect the current polarisation by applying an electric impulse to the

FeRAM cell. Different polarisations of atoms react differently to this voltage. If the cell has a `0` polarisation, it does not react. However, if the cell has a `1` polarisation, it responds by a measurable electrical impulse.

Endurance of FeRAM is high and comparable with DRAM. However, write operations are slow since they require changing of atom polarisation. Moreover, read operations are destructive and require the use of a regeneration mechanism to recover stored inside cells data.



(a) Logical element

(b) Encoding of bits

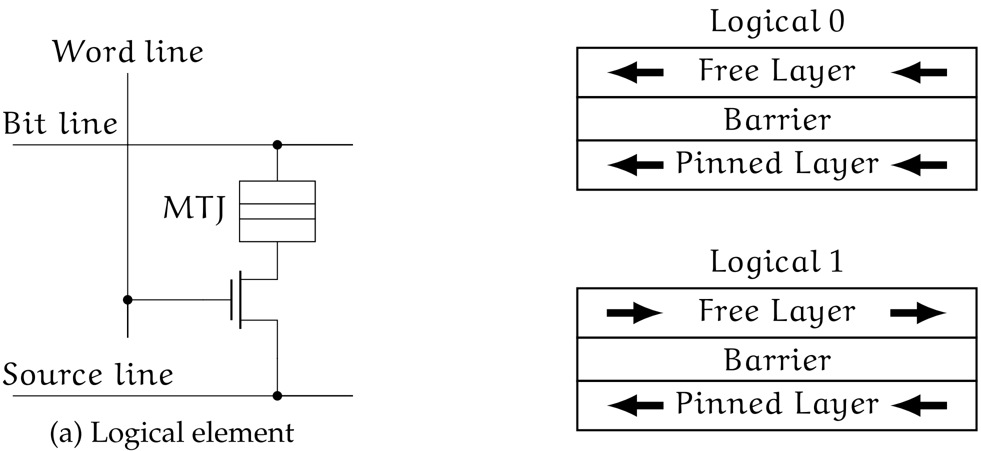
Figure 3.3: Ferroelectric RAM

3.1.1.4 Magnetoresistive RAM

Magnetoresistive memory is memory technology based on a phenomenon of *tunnel magnetoresistance* – a quantum mechanical phenomenon which occurs in a Magnetic Tunnel Junction (MTJ). An MTJ consists of three layers: top, medium and bottom (Figure 3.4b). The top and bottom layers are made from ferroelectrics, while the medium layer is a thin insulator [142]. One of the ferroelectric layers has a fixed magnetisation, while the magnetisation of the other layer can be changed externally. Parallel and antiparallel magnetisations of layers have explicit differences in resistance of the MTJ, and these are used to encode logical `1` and `0`.

A reading process in MRAM cells is based on a measurement of the magnetic resistance of the cell. A writing process is based on changing of the magnetisation of one of the layers by the application of an induced magnetic field (Figure 3.4a).

MRAM technologies develop rapidly, and the approach based on MTJ has evolved into other technologies, such as Spin Transfer Torque Magnetoresistive RAM



(a) Logical element

(b) Encoding of bits

Figure 3.4: Magnetoresistive RAM

(STT-MRAM) [142] and Thermally Assisted Switching MRAM (TAS-MRAM) [143]. A family of MRAM-related technologies is considered as the most likely for DRAM replacement, since the most of them are characterised by low read and write latencies, high density, and high endurance.

3.1.1.5 Comparison

Table 3.1 summarises the main characteristics of the candidate technologies and compares them with characteristics of conventional DRAM and NAND flash.

Table 3.1: Major characteristics of recent non-volatile memory technologies [1, 2]

Characteristic	**PC-RAM**	**FeRAM**	**MRAM**	**DRAM**	**NAND Flash**
non-volatile	yes	yes	yes	no	yes
Endurance	10^8	10^{12}	10^{12}	10^{15}	10^3
Write Latency	≈75 ns	100 ns	10 ns–20 ns	10 ns	10 µs
Read Latency	20 ns	70 ns	10 ns	10 ns	25 µs
Cell factor (F^2)	1–4	15–20	6–12	6–10	4

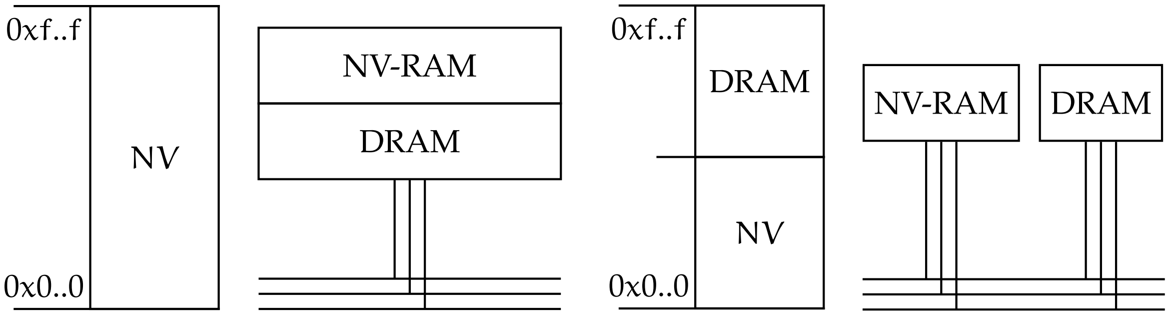
As can be seen, NV-RAM technologies have several differences compared to DRAM. Firstly, all of them have less endurance than DRAM. Endurance of FeRAM and MRAM is lower by three orders of magnitude, while endurance of PC-RAM is lower by seven orders of magnitude compared to DRAM. Secondly, DRAM has low and symmetric read/write latencies. However, the NV-RAM technologies have bigger values of latencies, and more importantly, all of them have asymmetry in read/write latencies. The most promising technology MRAM has similar read latency to DRAM,but write latency is still bigger [2]. Thirdly, the cell factor, which estimates the size of a cell independently from the semiconductor manufacturing processes way, shows that only PC-RAM have more significant density than DRAM. Cell factors of other NV-RAM technologies are comparable (MRAM), or bigger (FeRAM) than the cell factor of DRAM. Taking experimental technologies into account [144], one can estimate the realistic sizes of NV-RAM-based modules as comparable than those of modern SSD storage devices[33].

3.1.2 Related works: persistent systems

Different related works consider the architectures of persistent systems. One can separate these works into two groups. The first group of related works is devoted to the integration of NV-RAM into system architecture on a hardware level. The second group deals with the software aspects of persistent systems. Below, each will be considered independently.

3.1.2.1 Memory controllers

Hybrid architectures, i.e. architectures where both types of memory are used, is the central research area of this group of works. Figure 3.5 visualises two possible designs of hybrid architectures. The first figure (3.5a) depicts a generalised design introduced [145, 146] by Qureshi et al. In this scheme, DRAM and NV-RAM are used sequentially. Since low endurance and high read/write latencies characterise most NV-RAM technologies, a small DRAM cache attached to NV-RAM can mitigate these issues. The original work demonstrated that a small buffer (just 3% of NV-RAM's size) could bridge the latency gap between DRAM and NV-RAM.



(a) Sequential (b) Parallel

Figure 3.5: Possible approaches for NV-RAM integration

---

[33] For example, Intel announced 3D XPoint memory modules 512 GiB in size [29]

The second figure (3.5b) depicts a generalised design of hybrid systems [147] where both types of memory share the same memory controller. In this scheme, the memory controller plays a crucial role and becomes a very complex part of the hardware since it supports memory technologies of different types. For example, NV-RAM requires additional wear levelling, while DRAM does not. The system software should also distinguish between the different types of memory in its policies. As shown by FIRM [148] and NVM Duet [149], to effectively use NV-RAM, the system software should use different memory scheduling schemes for different types of memory.

On the one hand, this integration increases the complexity of hardware and software, but on the other, it enables the development of hybrid persistent systems which include volatile and non-volatile elements. In such systems, the system software can independently allocate volatile and non-volatile objects, thereby making part of the software persistent or not. The subsection 3.1.2.2 analyses such systems.

Approaches of wear levelling

As shown previously, NV-RAM technologies are characterised by low endurance. Thus, platforms with persistent memory may include mechanisms to increase dependability. Similarly to Error-correcting code (ECC) memory, NV-RAM could have built-in into a memory controller mechanisms of error detection and correction.

Schechter et al. demonstrated [150] that the Hamming-based ECC codes used in DRAM cannot be applied to NV-RAM technologies, in particular, to PC-RAM. In contrast to DRAM, NV-RAM is much less susceptible to transient faults, which usually protected by ECC. As a consequence, ECC-enabled persistent memory wears out faster than the cells it protects. As an alternative, the authors introduced Error-Correcting Pointers (ECP), error correction approach optimised for persistent memory.

Another approach, named Safer [151], was presented by Seong et al. Safer [151] is a multi-bit stuck-at fault error recovery scheme which demonstrated better lifetime improvement compared to ECP. Safer combines two approaches, the dynamic multi-group partitioning and the data block inversion. The former ensures that the partitioned groups include no more than one error bit, which can be detected and recovered by applying the data block inversion scheme. During this process, after an unsuccessful write operation, Safer performs the inversion write, and in a case of the second error, it detects a failure.

While Safer and ECP require custom NV-RAM devices, FREE-p [152], presented by Yoon et al., could be directly implemented inside the memory controller. This approach significantly decreased cost per bit for any type of persistent memory, but involved the OS kernel in the process of wear levelling. However, a letter work made by Qureshi [153] presented a more efficient and OS-independent approach.

3.1.2.2 Architectures of persistent systems

This group of related works considers future persistent platforms as hybrid architectures, where NV-RAM and DRAM co-exist as independent types of memory, or as fully persistent systems where only NV-RAM is used. These related works are usually devoted to the research questions: which parts of a system should be persistent and how to organise interaction of persistent software and volatile hardware. The related publications can be separated in accordance with a *model of persistence*, i.e. a generalised conception of how persistent entities are used in a system.

Because system architecture is the primary concern of this thesis, these models will be considered in detail. One can distinguish four major models of persistence.

Persistent filesystems

Filesystems organise data located in secondary storage. Since persistent memory can be considered as data storage, in-memory data can be organised in the same way as files on a hard drive. However, the programming of persistent memory differs from that of hard drives. As mentioned previously, NV-RAM offers byte-addressable access, which differs from block-addressable access of hard drives. Moreover, in-memory files can be directly accessed (so-called Direct Access (DAX) [154]), while storage data needs to be retrieved via DMA. These and other features of NV-RAM differentiate significantly classical filesystems from in-memory ones.

In-memory filesystems were pioneered together with technologies of BBRAM. The eNVy [155] storage system used a combination of volatile byte-addressable memory and non-volatile block-addressable flash as the main memory device. A battery-backed RAM module was integrated into the memory controller and worked as a cache. Later, Newton OS [39], Rio [136], and Conquest [156] used BBRAM in their architectures. Newton had a unified layer on top of the object-based storage system, which combined an optional flash device and main memory. Rio stored the file cache in a battery-backed memory module. Conquest, in turn, did the same with the filesystem metadata.

BPFS [157] was one of the first filesystems developed for NV-RAM. To ensure reliability, this filesystem used an improved technique of shadow copying [158]. In traditional shadow-paging file systems, updates to the file system initiate sequences of coarse-grained copy-on-write updates of the file system tree. Due to byte-addressability and the existence of fast, random write operations, in NV-RAM copy-on-write operations can be performed at finer granularity. As a consequence, BPFS can commit small changes at any level of the file system tree without updating the whole tree. While BPFS demonstrated almost twice the performance improvement compared to NTFS in ramdrive, this filesystem relied on multiple hardware mechanisms to enforce atomicity and ordering of write operations.

SCMFS [159] was integrated into the OS's memory management subsystem. Due to this, persistent files were available as contiguous regions in virtual address spaces of user processes. As a result, file access to SCMFS had a lightweight and straightforward interface. Moreover, SCMFS did not require any hardware modifications but relied on `MFENCE` and `CLFLUSH` x86 instructions to ensure atomicity and ordering of write operations. Additionally, this filesystem did not provide mechanisms for wear levelling.

PMFS [160] is another light-weight filesystem which bypassed the page cache and exploited the byte-addressability of NV-RAM. This filesystem used journaling to update small files, and shadow paging for big files. As with previous filesystems, PMFS required hardware-supported atomic and ordered memory writings.

Aerie [161] was not a real filesystem, but a substrate for the development of user-space NV-RAM-aware filesystems. Aerie had decentralised architecture, i.e. each user of a filesystem used its own user-mode library. However, privileged functions, which performed the mapping of physical pages into virtual address spaces, were implemented as a kernel driver and shared by all user-mode libraries. Aeries did not provide mechanisms for wear levelling and relied on `MFENCE` and `CLFLUSH` x86 instructions.

NOVA [162] was a log-structured filesystem which combined both types of memory, DRAM and NV-RAM. By design, NV-RAM was used to store log and file data. DRAM, in turn, was used to store radix trees, needed to perform search operations. NOVA was a DAX filesystem without wear levelling. Additional mechanisms which ensured durability were added in NOVA-Fortis [163] filesystem, which could detect and correct media errors.

Language- and library-based persistence

Persistent memory can be integrated into a user-space program as a new type of memory provided by a special memory allocator. This should manage the available persistent memory and within it, locate persistent objects.

Mnemosyne [40] enabled persistence for small objects which could be allocated statically or dynamically. By design, Mnemosyne used volatile memory (DRAM), persistent memory (NV-RAM), and secondary storage. The latter stores swap files used for the virtualisation of persistent memory. Three key services were provided by Mnemosyne: persistent memory regions, persistent primitives and durable memory transactions. The first service was used to store objects labelled as `pstatic`. The second enabled consistent updates, i.e. data modifications *without jeopardising correctness in the presence of failures* [40]. The third hid implementation details of persistent primitives by the compiler-provided transaction semantic.

NV-Heaps [41] used a memory region so-called *NV-Heap* to store persistent objects. As with Mnemosyne, NV-Heaps followed the hybrid design of a persistent system where NV-RAM and DRAM were used at the same time, but does not virtualise persistent memory, and provides automatic garbage collection and pointer safety. By request, the OS kernel mapped a chunk of persistent memory into the virtual address space of a program, and then, a special library provided allocations, locking and more inside the mapped region. NV-Heaps relied on architectural support developed for BPFS.

Several projects, such as Atlas [164], Makalu [165] and DNV Memory [35], shared the same persistence model as NV-Heaps and Mnemosyne, but concentrated on the durability of persistent objects. In Atlas, persistent objects accessed inside a Failure-Atomic SEction, which is the critical section extended by durability semantic, were protected from power failures. Makalu introduced an allocator which protected heap data from faults during the allocation phase and performed recovery of data together with control of memory leaks during the recovery phase. DNV Memory protected all persistent objects by reliable transactions based on software transactional memory.

Process-based persistence

All objects created inside a persistent process become persistent without the use of an additional library or allocator. However, the OS kernel needs to emulate persistent memory, or have NV-RAM in the system architecture.

The OS kernel can emulate persistence via checkpointing. KeyKOS [36], EROS [37] and Coyotos [38] implemented checkpointing of processes inside the paging subsystem. Each process, represented as a set of pages, had two versions: the current version, pages of which were distributed between the main memory and a swap file, and the checkpoint version, pages of which were stored on a disk. After a restart, the OS kernel booted from scratch, but then the recovery service retrieved previous processes from their checkpoints.

Grasshopper [166] introduced and implemented the conception of *orthogonal persistence* [167]. In contrast with the checkpoint-based persistence, where the system periodically makes a snapshot of processes and stores them on a disk, there is no disk as an independent entity in this model. All processes in Grasshopper had a unified, persistent memory layer which hid separation between primary and secondary storage.

NV-process [42] used a process as an abstraction for NV-RAM programming. Non-volatile memory was integrated as a new memory zone (like DMA) with its own allocator. Processes, allocated inside this memory zone, were decoupled from the whole operating system due to the use of independent mappings of virtual and physical pages. To achieve state-consistent execution, NV-processes used transactional execution mode.

System-wide persistence

The conception of process-based persistence can be generalised to the whole system. In this case, kernel components like device drivers additionally become persistent.

Whole-system persistence (WSP) [43] was introduced by Narayanan et al. In this project, all parts of a system were located inside NV-RAM. Volatile data of CPUs, such as caches and registers set, were protected against loss by the *flush-on-fail* mechanism. This mechanism required a POD and initiated a sequence of data flushes in a moment of power failure. Other types of failure, such as crashes of software caused by software errors, and system restarts are processed in an ordinary way since the system architecture includes conventional secondary storage and does not change the life cycle of programs.

In contrast with process-based persistence, system-wide persistence needs to take into account states of devices and persistent drivers. After the recovery of a persistent system, volatile devices need to be re-initialised and re-configured, i.e. turned into the coherent with persistent drivers states [31]. For example, at boot, a network driver uploads a new MAC address into a device, but after a power failure, this value will be lost and needed to be uploaded again. WSP uses the suspend/resume interface of drivers for this purpose, which in the end, transforms all power outages into ″suspend/resume″ events [43]. This process requires a significant amount of residual energy provided by the corresponding hardware, because saving and restoring device state takes much longer than the flushing of CPU caches.

3.1.3 Conclusion

As can be seen, there are many different approaches for integration of persistent memory into conventional systems. From the hardware point of view, persistent systems can be homogenous, i.e. only include NV-RAM, or heterogeneous (hybrid), i.e. combine both NV-RAM and DRAM. From the software point of view, NV-RAM can be introduced at different abstraction layers: a variable, a process, an operating system, a file. Table 3.2 summarises these *models of persistence*.

Table 3.2: Comparison of NV-RAM integration abstractions

Type of Persistence	Memory Connection	NV-RAM Abstraction	Modification
Language/library	Hybrid, Parallel	Variables and objects	Kernel, programs
Process	Hybrid, Parallel	Whole programs	Kernel
System-wide	NV-RAM	Programs and kernel	Kernel
File systems	Hybrid or NV-RAM	Files	Kernel, drivers

All of these approaches need the support of persistent memory inside the kernel, and some of them additionally require modification of applications. Even persistent file systems, which do not provide persistence for execution context and in general can be implemented as an independent driver, still, need the support of NV-RAM inside the kernel [160]. In other words, despite the different hardware platforms and abstraction layers, all these have one common drawback: legacy systems cannot profit from the introduction of NV-RAM. This drawback is addressed in the following section, which introduces a model of persistence for legacy and proprietary programs.

3.2 NV-Hypervisor

Despite the fact that the technologies of persistent memory are in the early development stages, some of them, for example BBRAM, can be used now. These technologies are interesting on a practical level because they repeat the behaviour of future NV-RAMs without the drawbacks like asymmetry in read/write latencies or low durability (section 3.1.1.5). Moreover, these technologies are market available now and can be applied to existing systems to solve essential challenges like the prevention of power outages of cloud infrastructures.

In accordance with studies made by Ponemon institute [168, 169, 14], the most common failures in data centres are related to power failures. The average cost of a data centre outage rose from $505,503 in 2011 to $690,204 in 2013, and then up to $740,357 in 2016. These costs include not only direct losses of revenue caused by downtime of services, but also indirect losses, such as temporary degradation of service quality followed by the outage [168].

In-memory computing relies on the use of in-memory runtime data, temporary elements of which are cached in the main memory. An example of such data can be filesystem caches created by a DB during the work. If the data is not secured in persistent storage, the DB needs to recover it from a persistent log after a restart. However, even if the data is secured, after the restarting, the DB needs time to populate its caches to reach the previous performance.

Multiple techniques were introduced by research communities to mitigate these issues; logging and checkpointing [170] being the most common ones. These mechanisms require additional resources, have an impact on performance during normal operations, and delay recovery [171]. Another common approach is to use protective technologies which specifically address power outages. UPS can prevent the loss of runtime data and perform mirroring of memory content to storage during the power failure. However, these devices are expensive, fragile, and require periodic maintenance (because of battery degradation). As a result, the devices are the primary root cause of power outages [168].

Persistent memory can retain in-memory data without an external source of power. Allocated inside, persistent memory runtime and application data can survive power outages and do not require long recovery processes. This can decrease downtime duration and performance recovery time.

As was shown in section 3.1, technologies of persistent memory have a significant impact on system architecture. However, for users of cloud services, this integration should be seamless and transparent. Cloud services rely on code reuse and legacy software – fundamental software development practices which reduce time-to-market.

Unfortunately, no one of the described models of persistence is entirely suitable for cloud services (section 3.1.3). For example, persistent filesystems do not provide persistent execution contexts. Language- and library-based persistence requires modification of the whole software stack and thus cannot be applied to proprietary/legacy software. Process-based persistence also requires modification of the kernel and does not provide persistence for the kernel components, for example the page cache. This cache consists of storage pages which will be lost in the case of a power failure. System-wide persistence requires modification of the kernel, but also relies on a significant amount of residual energy used to save the state of drivers in a moment of power failure. This requires the use of additional hardware to perform complex procedures of drivers suspending.

To make persistent memory "invisible" for users of cloud services, it should be offered at a system layer which is transparent for the customer's software. The virtualisation layer provided by a hypervisor is the best-suited layer for this purpose. The hypervisor virtualises memory management of Virtual Machines (VMs) and thus can introduce persistent memory without affecting the guest VMs. Moreover, since the hypervisor virtualises devices, the system software of VMs should not be modified to maintain state consistency of devices and drivers. Thus, *hypervisor-based* persistence, offered by a NV-RAM-aware hypervisor, can provide transparent persistence for cloud services.

This section is devoted to NV-Hypervisor: a hypervisor which was built in accordance with the conception of hypervisor-based persistence. Subsection 3.2.1 describes architecture of NV-Hypervisor. Subsection 3.2.3 discusses implementation details of NV-Hypervisor. Subsection 3.2.4 evaluates the hypervisor. The following subsection describes related works while the subsection 3.2.6 summarises the section.

3.2.1 Design and Architecture

NV-Hypervisor inherits the **parallel memory model** of hybrid persistent systems (section 3.1.2.1). In this model, both types of memory technologies are presented in the system architecture. The OS kernel is located inside DRAM and can independently access volatile and non-volatile memory. Accordingly, the kernel can allocate volatile or non-volatile memory for applications.

NV-Hypervisor inherits the **flush-on-fail** approach to flush transient CPU state to NV-RAM at the moment of a power failure (section 3.1.2.2). It is impossible without an early detection of the power failure, and a POD is a hardware component which adds this functionality. A POD continuously monitors the input voltage and immediately sends a signal to the system when the voltage drops below a defined threshold. On detection of a power failure, residual energy, i.e. all the energy accumulated in the electrical circuits, is used to store the *virtual context* of VMs into NV-RAM. In the remainder of this section, this process is referred to as *fixation*.

NV-Hypervisor relies on **Battery-backed RAM**. This type of persistent memory has the same characteristics as DRAM, and as a result, the hypervisor does not use any approach for wear levelling[34]. However, NV-Hypervisor virtualises persistent memory, since the capacity of BBRAM is much less than that of prospective NV-RAM technologies[35].

3.2.1.1 System architecture

Figure 3.6 depicts system architecture of NV-Hypervisor. As can be seen, NV-Hypervisor has a multilayer structure which includes hardware components, software components, and inter-layer communication mechanisms.

The hardware platform of NV-Hypervisor is based on a commodity server platform with general-purpose CPUs, volatile DRAM as the main memory, and secondary storage. Additionally, the platform includes two new devices: NV-RAM in the form of BBRAM, and a Power Outage Detector (POD).

The software components of the system are based on a general-purpose OS. The OS kernel includes two additional components: a POD driver and a NV-RAM driver. The former manages interrupts from a POD, while the latter provides support for NV-RAM.

---

[34] Alternative to BBRAM can be STT-MRAM, characteristics of which are very close to that of DRAM

[35] See section 3.1.1.5 and note 33

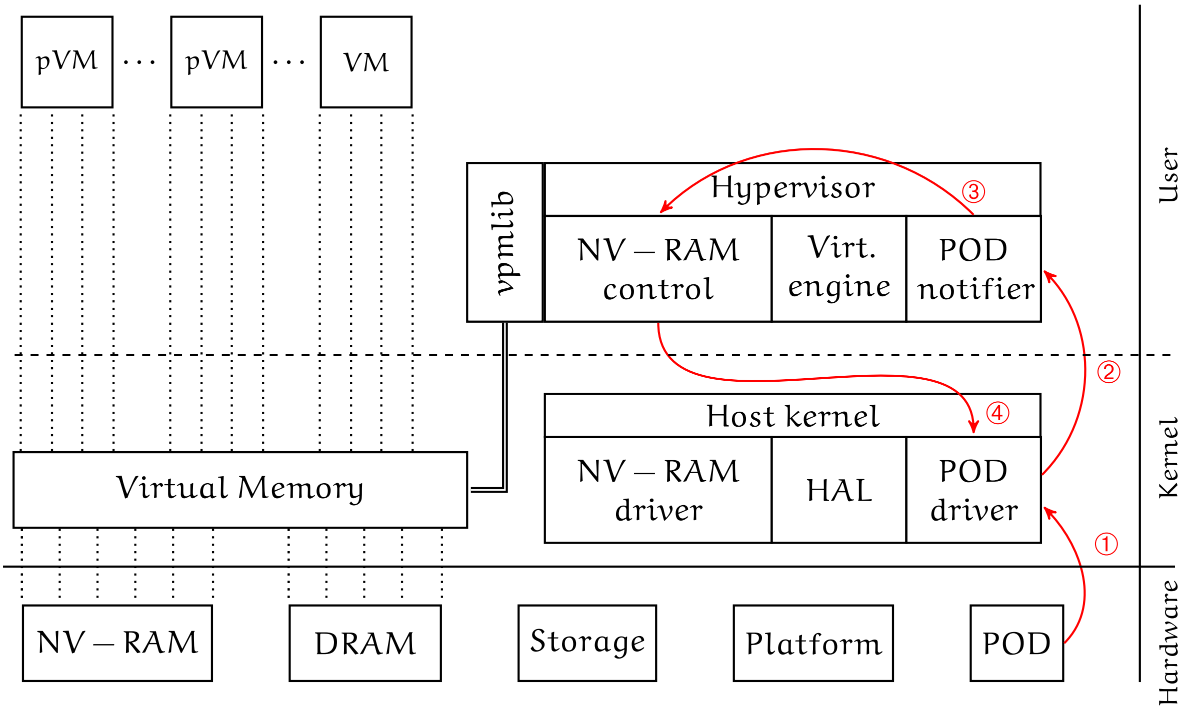


Figure 3.6: Architecture of NV-Hypervisor

NV-Hypervisor runs on top of this software stack as an ordinary volatile process. It manages all ordinary and Persistent Virtual Machines (pVMs). In addition to the typical components of a hypervisor, NV-Hypervisor also includes a NV-RAM control module and *vpmlib*. The hypervisor provides virtualisation of persistent memory by managing mappings between the virtual and physical addresses of pVMs using these two components. The third component, named POD notifier, is used in the *fixation* process.

In the case of a power outage, all components work together to save the pVMs reliably. As shown in Figure 3.6, the POD first detects an incoming outage and issues (①) an interrupt to the POD driver. The driver forwards (②) it to the POD-Notifier located in NV-Hypervisor. The POD notifier performs the saving of the pVMs's virtual context (③) and then notifies the POD driver about the end of the fixation process (④). Afterwards, the kernel flushes all caches and stops memory operations.

3.2.2 Persistent Virtual Machines

The minimum persistent executable unit in NV-Hypervisor is a VM. This can be represented as a combination of two components. The first component is a RAM image of the VM. This image is a chunk of host memory which is used for emulation of guest system RAM. The image includes all runtime data of guest services such as the kernel and programs. The second component is *virtual context* (or just *context*), which includes states of VM's virtual CPUs and emulating devices. The devices (and the virtual CPUs) are parts of the hypervisor, but can be exported or imported.

Hypervisor-based persistence implemented in NV-Hypervisor is *memory-centric*. It means that only a RAM image is permanently located inside persistent memory. All other components do not use persistent memory at all, or only partly. The host system, its device drivers and NV-Hypervisor itself stay volatile and shall be restarted after every failure. Each virtual context is located in the volatile memory (since it is part of the hypervisor), but the hypervisor has a special serialisation mechanism which saves this data to NV-RAM at the moment of a power failure. This combination of volatile and non-volatile components keeps the whole software stack and the hardware platform coherent.

Following this approach, the recovery of a pVM, in essence, is performed in the same way as the recovery of an ordinary VM from its snapshot. In other words, to recover a pVM, NV-Hypervisor needs to create an empty context of a VM, attach the persistent RAM drive to this context, and then apply the previous stored virtual context to this VM. After this procedure, the pVM will be retrieved and ready to continue execution on its own.

3.2.3 Implementation

NV-Hypervisor extends the commonly used virtualisation platform QEMU. As a hardware platform, the prototype of NV-Hypervisor uses the NVDIMMs-based solution provided by Viking Technology [138]. The NVDIMMs are implemented as DRAM memory modules which are backed with NAND memory of the same size. The NVDIMMs have a supercapacitor, which is used as a source of power at the moment of a power failure to mirror the DRAM state to flash memory.

3.2.3.1 Core services

The detection of a power failure is implemented by a POD, which comes attached with the NVDIMMs. Communication between the POD notifier and the POD driver is implemented by a blocking `ioctl`-syscall. For that, NV-Hypervisor first spawns a pthread for a POD notifier and then, the notifier issues a special system call. The POD driver blocks this call inside the kernel until a power outage is detected.

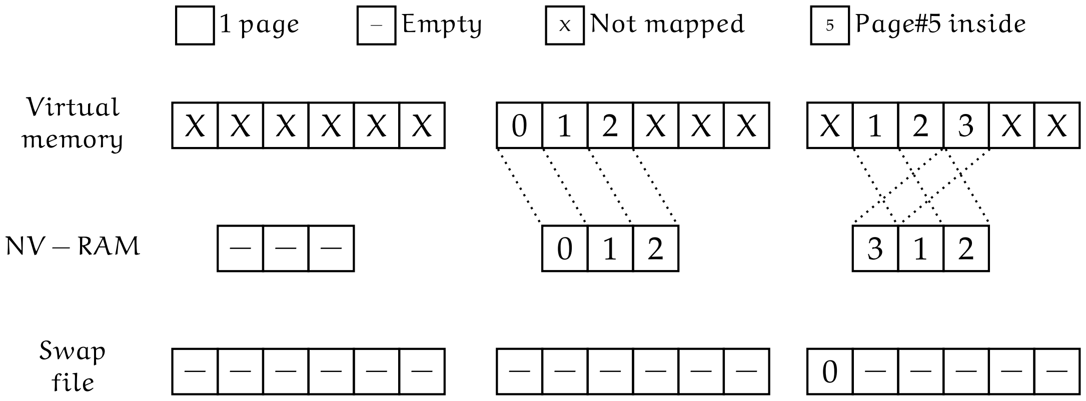
Two additional QEMU monitor commands were implemented to manage pVMs: `dump-devices` and `nv-restore`. The first command serialises and saves the *virtual context* of a pVM, while the second applies the previously saved context to a pVM. Also, two additional command line arguments were added to QEMU: `-nv-restore` and `-nvm`. The first flag forces QEMU to perform the `nv-restore` command immediately after the start, while the second flag forces QEMU to use *vpmlib* for memory management.

3.2.3.2 Virtual Persistent Memory

NV-Hypervisor uses a *vpmlib* library for persistent memory management. The library is implemented in user-space, offloads persistent memory-related policies from the kernel and performs two functions. Firstly, it provides concurrent primitives for persistent memory allocation/deallocation. Secondly, it extends available volumes of persistent memory by the use of the virtualisation layer together with secondary storage. As a result, multiple instances of NV-Hypervisor, or any other NV-RAM-aware programs, can allocate and use bigger than available chunks of persistent memory.

The *vpmlib* library provides a minimalistic API to allocate (`nvalloc()`) and free (`nvfree()`) regions of Virtual Persistent Memory (VPM). Internally, this is implemented by the system calls `mmap` and `munmap` to map pages from the persistent memory (located in `/dev/mem`) into the virtual memory of the running process. In between these levels, *vpmlib* manages persistent memory, i.e. provides name-based allocation, implements the swapping mechanism and recovery of previous allocated regions. The latter is possible due to the allocation inside NV-RAM of all structures which describe configurations of persistent regions.

Despite the simple API, *vpmlib* covers many technical aspects of memory management, which are discussed below.



(a) Initial state (b) Full NV-RAM (c) Swapped out page

Figure 3.7: Simplified example of a virtualised persistent memory region

Concurrency or allocations

Multiple instances of NV-Hypervisor should have the ability to allocate/free memory regions concurrently. Because each instance has its own version of the policy library, i.e. *vpmlib*, all instances should have a mechanism for data access serialisation. The *vpmlib* library ensures this through the mechanism of filesystem locks provided by the Linux kernel.

Page allocation and swapping

Each instance of NV-Hypervisor manages a single chunk of persistent memory. If the size of a pVM is equal to or less than the size of the chunk, the *vpmlib* library does not virtualise persistent memory. Instead, it maps the requested size of its own NV-RAM chunk to an empty region of the NV-Hypervisor's virtual address space and uses this region to store the RAM image of pVM. However, if the requested size of pVM is bigger than the size of the available memory, the *vpmlib* library uses the available persistent memory as a cache for frequently used persistent pages, and stores rarely used pages in a swap file.

At the beginning, the *vpmlib* library allocates a swap file and a range of virtual memory inside NV-Hypervisor's address space. The size of the range is equal to the requested size of pVM. Moreover, the sum of the swap file size and the NV-RAM chunk size is also equal to the requested size of pVM. Pages of the allocated virtual memory do not have corresponding physical pages, and any access to them causes a `SIGSEGV` page fault signal. The *vpmlib* library registers a signal handler `vpm_fault_handler`, zeroes the

NV-RAM chunk and the swap file, clears Persistent Page Table (PPT), which describes mappings of pages, and finalises the initialisation process.

The initial state of all components is presented in Figure 3.7a. As can be seen, a pVM has six pages, but only three of them can be located inside NV-RAM in this example. After the initialisation, all pages inside the swap file and the NV-RAM are empty, and there are no mappings between NV-RAM and the virtual memory.

Any access by the virtualisation engine of NV-Hypervisor to any page of the reserved range causes a page fault, which is processed by the registered page fault handler `vpm_fault_handler`. The page fault handler receives an address of a virtual page, access to which caused the page fault, and computes the number of the NV-RAM page which needs to be mapped to this address. Next, the handler maps an empty page of NV-RAM to the fault address, copies data from the corresponding offset of the swap file into this page, modifies PPT, and resumes the execution. In the considered example (Figure 3.7b), the page fault handler detects three access attempts to pages `#0`, `#1` and `#2` and maps NV-RAM pages to the corresponding addresses.

However, at some moment, all pages inside NV-RAM become used, and the page fault handler should perform *swapping*. During this process, the handler first identifies the oldest allocated page, moves its data to the swap file, flushes storage caches, and atomically updates PPT. Then, the handler retrieves content of the requested page from the swap file, copies it into the previous freed page of NV-RAM, and atomically updates PPT the second time. Then, the handler changes mappings and resumes the execution. Figure 3.7c visualises such case. As shown, during the swapping process, page `#0` was moved into the swap, while page `#3` occupied the freed place. Pages `#1` and `#2` use the same mappings, while pages `0`, `4`, and `5` are unmapped.

Name-based allocation

During the recovery process, NV-Hypervisor creates an empty VM, attaches a RAM drive located inside NV-RAM and applies previously saved *virtual context*. After that, the VM can continue own execution from the same state which it had before the fault. If a host system has more than one NV-Hypervisor instance, then it needs to have a mechanism to distinguish the previously assigned RAM image. This is done by a name-based allocator, which is provided by the `nvalloc()` function.

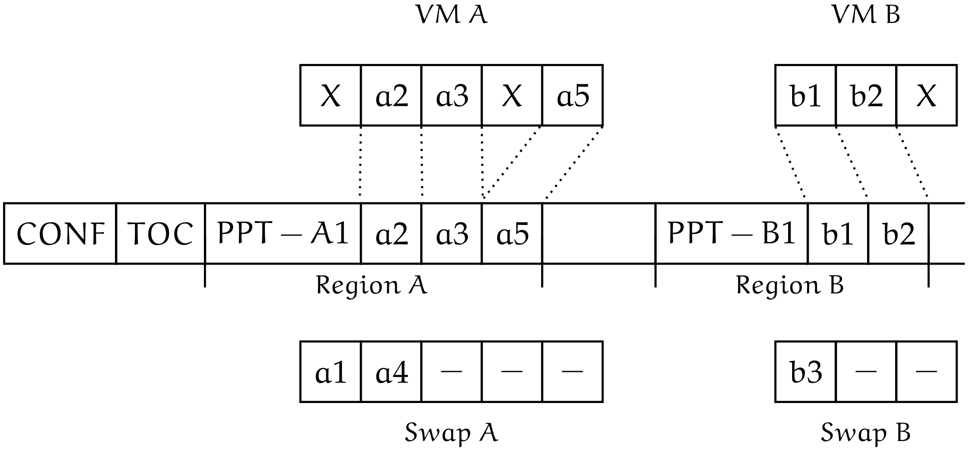


Figure 3.8: Name-based allocation of two VMs

In contrast to the ordinary malloc function, this function receives two arguments. One of them is a unique identifier of a memory chunk, the second one is the size of the requested memory chunk. This identifier is computed as a hash value created from the pVM's name and an allocation sequence number[36].

Figure 3.8 visualises relationships between the name-based allocator and the virtual memory engine. As can be seen, NV-RAM has several regions. The first one, named CONFiguration section (CONF), consists of technical information like a location of the synchronisation primitive, page sizes and more. The second part is a Table of Contents (ToC). The ToC is a simple table, elements of which are encoded associations between identifiers of memory regions with their sizes and offsets. The remaining part of NV-RAM is occupied by allocated regions or empty blocks.

Each allocated region also has a small header, named Persistent Page Table (PPT). This header describes mappings of persistent pages of this region as well as a configuration of the corresponding swap file. Each NV-Hypervisor instance manages only pages located in the assigned memory region. As a consequence, the synchronisation primitive is used only for rare moments of allocation and freeing of regions and does not impact the performance.

---

[36] NV-Hypervisor has a single threaded design and thus, sequences of allocations are always the same inside each hypervisor instance.

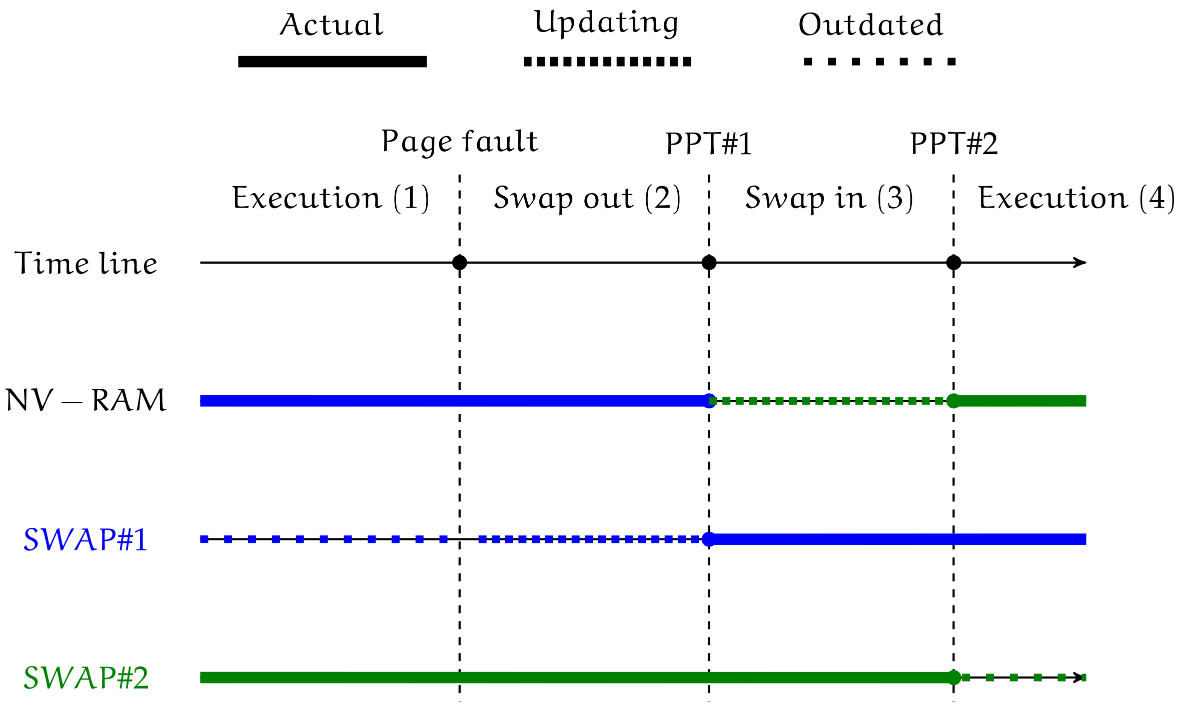


Figure 3.9: States of the virtual memory engine during a page fault

Critical path

The page swapping process is *failure-atomic*, i.e. memory pages are recoverable independently at the moment of a power outage. This is possible because of the independent updates of PPT.

The virtual memory engine have several states, which are depicted in Figure 3.9. The engine can be in one of four states when a page fault happens: *Execution* (1), *Swap out* (2), *Swap in* (3), and *Execution* (4). A page fault transforms the engine from (1) to (2), while the first update of PPT transforms from (2) to (3). The second update of PPT transforms the engine to state (4).

States (1) and (4) are safe because there is no data exchange between a swap file and persistent memory. States (2) and (3) are also safe because these operations will be restarted after the power outage recovery: thanks to the separated updates of PPT, the swapping in and out processes are not done until this is not reflected in PPT.

Finally, there are two moments which could be vulnerable to faults: the updates of PPT. However, these updates are very short (just a few 64-bit values), and the engine always finalises these operations in the fixation process. Consequently, all operations of the virtual engine are *failure-atomic*.

3.2.4 Evaluation

The following questions were addressed in the evaluation of NV-Hypervisor:

- How fast is the fixation process? E.g. what are the demands in residual energy?
- Does VPM influence runtime performance? E.g. due to swapping of pages.
- Does NV-Hypervisor decrease recovery time?

A server platform from Viking Technology equipped with two six-core Xeon CPUs, 4 GiB of DRAM and 4 GiB of NVDIMMs was used in the evaluation. The host operating system was based on Fedora 17 with the Linux kernel version 3.4.12. The same version of Fedora was used as a guest OS inside all VMs. QEMU version 1.4.2 was used as a virtualisation engine.

3.2.4.1 Time to fixate a pVM

As described earlier, residual energy is used as a source of power for the *fixation* process of pVMs. Different power supply units and different implementations of PODs provide different amounts energy. The precise measurement of time necessary to fixate a pVM can help formulate the requirements for new power devices.

Table 3.3: Timings of the pVM fixation process

Step	**Time, ms**
① **Interrupt delivery**	n/a
② **NV-Hypervisor notification**	< 1
③ **Fixation**	< 1
④ **Return to the kernel**	< 1
Total reaction time	< 3

Table 3.3 shows timings of a pVM saving process. As can be seen, each step of the saving process, i.e. delivery of a signal from a POD driver to NV-Hypervisor, *fixation*, returning to the kernel, does not exceed 1 ms each or 3 ms in total. This also was tested with up to four pVMs without any significant difference in the *fixation* time. This is not surprising since for every pVM, only a very small amount of data needs to be copied to the NV-RAM.

Two factors play a crucial role in the fixation process. Firstly, the size of CPU caches defines the volume of data which needs to be fixated. The number of pVMs does not impact this and this time is constant for any number of virtual machines. Secondly, the fixation process also includes the serialisation process when NV-Hypervisor exports states of virtual devices to NV-RAM. This process requires time, but different instances of NV-Hypervisor can do this in parallel. Since the evaluation platform has four cores, the fixation time of four pVMs is very close to that of a single pVM.

3.2.4.2 VPM microbenchmark

The Redis Key-Value Store (KVS) was used to evaluate the performance impact of the VPM implementation. The *redis-benchmark* tool, which is provided by the Redis project, was used as a load generator. The generator and testing services were located on different servers connected by 1 GiB Ethernet.

Firstly, the performances of pVM and volatile VM were compared. For that, two virtual machines with the same software inside, but different memory types were created and benchmarked. In both cases, the size of memory available to a VM was limited by 256 MiB. As expected, both types of VMs demonstrated the same performance over multiple experiments, since NVDIMMs were based on conventional DRAM modules. Moreover, the parallel execution of multiple pVMs did not impact the performance. This was not surprising since VPM can perform mapping and accessing of persistent pages in parallel. The synchronisation is only required when new regions of persistent memory need to be allocated, which happens only on pVM startup.

Secondly, the performance of pVMs with different sizes of available physical regions were compared. The same benchmark was repeated several times, and the testing pVM had different share between sizes of available physical and virtual memories each time. The virtual memory size was always constant (256 MiB), while the size of available NV-RAM decreased in each experiment by 25.6 MiB (10%), starting from 256 MiB (100%) down to 52.2 MiB (20%).

Figure 3.10 shows results of the benchmarks. The X axis represents Memory Residence Ratio (MRR) computed as:

$$MRR = \frac{InMemory}{InMemory + InSwap} * 100\%$$

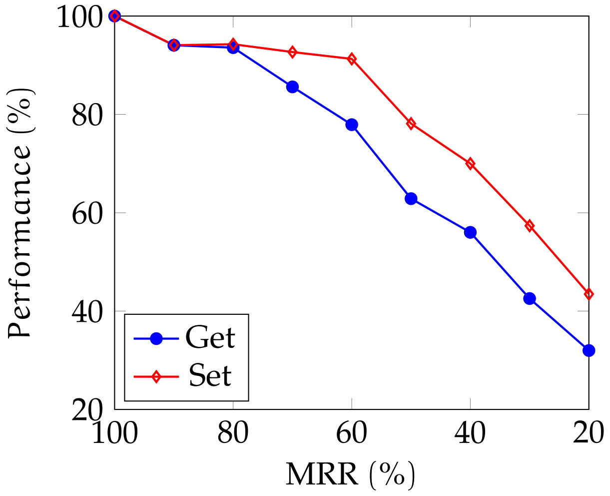


Figure 3.10: Redis performance degradation for set/get requests as a function of MRR

where $InMemory$ is a size of available NV-RAM, while $InMemory + InSwap$ is a size of virtual memory. The Y axis represents the relative performance of a benchmarking setup, compared to the baseline (100%).

As can be seen, without swapping (MRR=100%), the pVM performs at the same speed as a usual VM, both for *get* and *set* requests. However, by decreasing the MRR, the performance drops almost linearly for the *get* requests, while the performance of *set* requests stays at 90% until the MRR reaches 60%.

The reason for the difference in behaviour is hidden in the memory access patterns used in the benchmarks. During the *set* request benchmark, new memory is allocated linearly and VPM is not involved until consumed memory reaches $InMemory$ border, i.e., with a MRR of 90% the system will have to swap-out pages only when the amount of free memory falls below 10%. In contrast to this, the benchmark for *get* requests performs random *get* requests on a pre-filled database. Therefore, the probability for *get* requests to access a swapped out page is very linear to the MRR.

Figure 3.11 supports this hypothesis. The figure visualises a page fault rate for different benchmarks and different MRR values. As can be seen, in the case of the *get* benchmark, for both values of MRRs (40% and 60%), pVMs demonstrate different, but an almost constant page fault rate. Both *set* benchmarks, in turn, have nearly the same page fault rate during the first 100 seconds. An average value of this rate is significantly lower than that of the *get* benchmarks. However, both *set* tests have points when the rate of page faults starts growing rapidly (and linearly), and the test with the smaller MRR

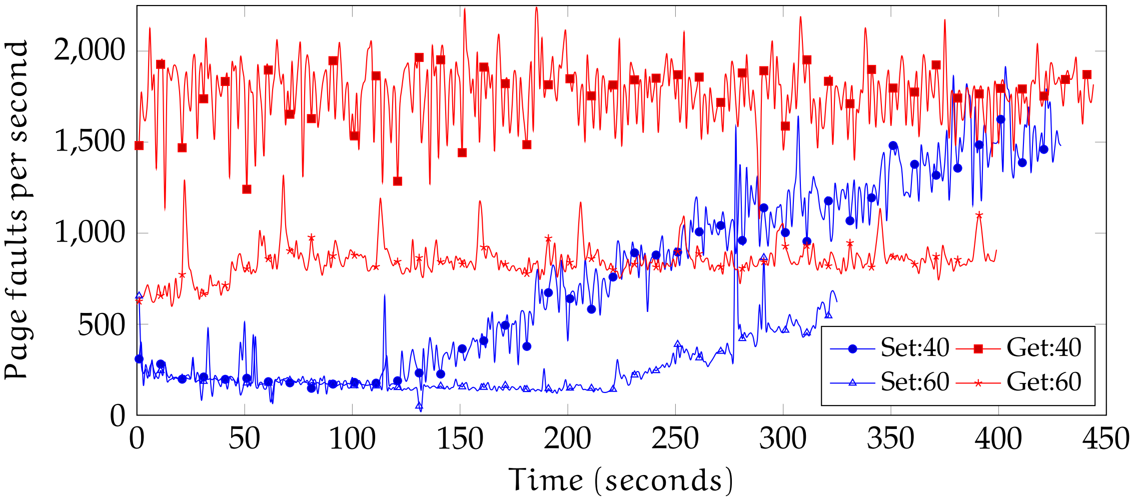


Figure 3.11: Number of page faults for different MRR (fixed workload)

(40%) reaches this point early. In other words, the diagram supports the hypothesis that the *get* benchmarks constantly experience page faults since they randomly access memory and hit swapped out pages, while the *set* benchmarks start experiencing page faults when the content of an in-memory DB exceeds the $InMemory$ value.

In sum, the benchmarks expectably demonstrated that the virtualisation of persistent memory linearly impacts the performance of virtual machines, but the exact impact depends on memory management strategies of a virtual service.

3.2.4.3 Recovery of a cloud service

Recovery behaviour of NV-Hypervisor was evaluated by a pVM with a memory-heavy service inside. A MySQL database was used as the service, and *sysbench oltp test suite*[37] was used as a load generator. An evaluation scenario was based on the simple power failure case: a hardware platform gets a power outage at a random moment of time, and then, after the recovery of power, it needs to restart services independently. The evaluation compares the recovery behaviour of an unmodified vanilla GNU/Linux system running plain QEMU with the NV-Hypervisor prototype.

Table 3.4 details the different phases during the boot process. As shown, the actual boot process of the host operating system is quite similar. In fact, the NV-Hypervisor-based system is even slower as the NVDIMMs have to be initialised and checked at the start. The Viking platform used for evaluation was an early evaluation

---

[37] `http://sysbench.sourceforge.net/`

Table 3.4: Boot process comparison

Boot phase	**Commodity system (sec)**	**NV-Hypervisor (sec)**
DB warm up	566	n/a
DB recovery	54	n/a
GuestOS boot	31	n/a
QEMU start	0.2	8
Host boot	108	108
BIOS	15	15
NVDIMMs init	n/a	109
Server boot	36	36
$\sum$	**810.2**	**276**

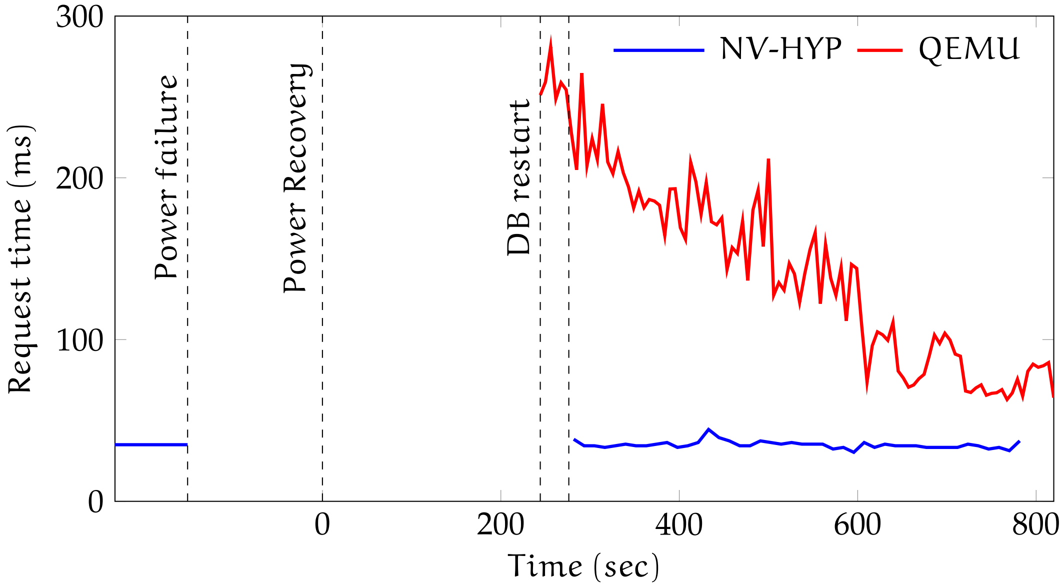


Figure 3.12: Process of a DB performance recovery

platform, and future NV-RAM-based platforms will probably not need these steps. After QEMU/NV-Hypervisor is running, the situation changes as for the commodity system: the VM and its services have to be started, while in the case of NV-Hypervisor this is not necessary. Still, up to this point, the NV-Hypervisor-based solution is ≈13% slower.

However, the picture changes dramatically once the actual runtime behaviour of both implementations is taken into account. A relational database typically has a long warm-up phase until queries can be answered at full speed. Accordingly, the time until booth settings were fully operation was measured. For that, the *sysbench* utility was used, which creates a table with 1.000.000 lines and measures request response time for a random query applied to this table. The warm-up phase took 566 seconds for the commodity system, and zero time for the NV-Hypervisor-based solution.

Figure 3.12 visualises the warm-up process. At the beginning, both systems demonstrate similar performance. Then, a power outage takes place, and a recovery process begins at time zero. While the commodity solution continues operation after 244.2 seconds, the NV-Hypervisor-based requires 31.8 additional seconds. However, initial queries of the commodity system processed immediately after the recovery require about a factor of 5 more time, compared to the NV-Hypervisor-based instance.

In sum, the evaluation demonstrated that NV-Hypervisor has a constant time for any VM recovery, which heavily depends on hardware support of NV-RAM. When taking the actual service response time into account, the commodity system demands for a factor of 2.9 longer until the recovery is fully finished.

3.2.5 Related works

In addition to the works described in Section 3.1, there are several other related works. Rapilog [172], used residual energy to store transaction logs of a database at the moment of a power failure. NV-Hypervisor uses the same approach in the *fixation*.

Kannan et al. presented [173] a virtualisation engine for persistent memory. NV-Hypervisor and this work target the same issue but resolve it differently. NV-Hypervisor provides virtualisation at the level of the hypervisor, while Kannan et al. provide the same inside the kernel. While the kernel-based approach can be used instead of the hypervisor-based one, the latter enables more fine-grained management of resources, for example, assigning of the VM images as a swap file.

Ex-Tmem [174] also use persistent memory together with virtual machines. Here, persistent memory was used as a fast caching system for swapped out pages, i.e. instead of moving swapping out pages to secondary storage, the virtualisation engine moved them to persistent memory. This approach significantly improves performance during runtime, but does not provide persistence to virtual machines and does not decrease the performance recovery time.

Mini-Ckpts [175] introduced a system recovery approach in which programs located inside persistent memory can tolerate crashes of the OS kernel. This approach has many similarities with NV-Hypervisor, but NV-Hypervisor and Mini-Ckpts have different fault models: NV-Hypervisor protects virtual machines from power outages, while Mini-Ckpts does not. However, Mini-Ckpts can be applied to pVMs.

Several works addressed the consistency issue of persistent drivers and volatile devices. Whole-system persistence [43] suspends persistent drivers in a moment of power failure. This process requires a significant amount of residual energy. Ohmura et al. proposed [176] another approach: all write operations to device configuration registers should be logged into persistent memory. During the recovery phase, the logged operations should be performed again, which turns a reset device into a pre-failure state. This approach requires significant modification of drivers. NV-Hypervisor uses a virtualisation layer to decouple volatile devices and persistent software. It saves only the volatile context, the size of which is much less than that of the kernel drivers, and requires modification of neither host nor guest drivers.

Continues VM replication, presented by Remus [171], Kemari [177], qemu-mc [178], is another approach for fast recovery of virtual machines. This approach uses two servers, one with a replicated VM, the other with a backup VM. The replication system keeps these virtual machines in a synchronous state, and if the first VM or the first server crashes, the second VM continues execution. To minimise the performance impact of the replication system, these servers should be located inside the same data centre, and thus, in case of a datacenter-wide power outage, both servers will lose their state, which is not the case with NV-Hypervisor. At the same time, NV-Hypervisor does not protect virtual machines from crashes, and to prevent this, the hypervisor can be extended by one of these continuous replication systems.

3.2.6 Summary

Power outages in Infrastructure-as-a-Service clouds may cause a loss of data. Additionally, the performance of the services is often degraded after a system restart. NV-RAM can retain stored information without an external source of power and, thus, can secure in-memory data. However, integration of persistent memory requires the revision of existing architectures. Existing approaches cannot be applied since they require significant modification of system and user software. NV-Hypervisor, developed in accordance with the *hypervisor-based* model of persistence, provides transparent persistence for legacy and proprietary software. It supports virtualisation of persistent memory, parallel execution of multiple pVMs and, as was shown during evaluation, can significantly decrease the performance recovery time of a memory-heavy service.

4. Conclusion

Scalability and security are the two important challenges which impact further development and propagation of clouds. Two research areas address these issues. Persistent systems based on modern NV-RAM technologies can significantly increase performance of cloud systems. Trusted execution, supported by the modern CPU extensions like AMD SEV and Intel SGX, enable trusted computations in untrusted environments. However, the CPU extensions and NV-RAM impact system architecture and require additional system support. This thesis addresses that through two research questions:

Q1: What is a better programming model and system support for SGX enclaves?

In section 2.1 I provided a detailed analysis of Intel SGX and its shortcomings. I showed that while SGX enclaves protect user-space programs, they do this with performance penalties because of the high transition costs and the EPC paging.

***E*Actors** As the first contribution, I developed the *E*Actors programming framework for SGX enclaves.

In section 2.2 I demonstrated that existing programming approaches and frameworks do not fully use the advantages of enclaves: the Intel SDK offers only basic primitives while frameworks like SCONE, Haven, and Graphene-SGX offer support for monolithical single enclave applications. In contrast to these, the *E*Actors framework offers a lean actor programming model, flexible deployment based on multiple enclaves and high-performance enclave-to-enclave communication primitives. In the evaluation, I demonstrated how a commodity service built on top of this framework can benefit from multi-enclave design and outperform even non-enclaved service with the same functionality.

STANlite As the second contribution, I developed the low footprint enclaved database engine STANlite.

In section 2.3, I demonstrated that the existing hardware-based paging mechanism significantly decreases the performance of memory-consuming applications, in particular databases. As an alternative I developed a software-based paging mechanism, which performs the page swapping without involvement of the untrusted system software, and developed a database engine STANlite on top of it. STANlite has minimalistic TCB, features the software-based virtual memory engine with various policies and the high-performance ECALL-free communication layer. In the evaluation, I demonstrated that the combination of these two approaches allows STANlite to significantly outperform the enclaved baseline for databases bigger than the EPC size.

Q2: What is a better model of persistence and system support for persistent memory?

In section 3.1, I presented the overview of prospective persistent memory technologies and an analysis of their features and limitations. I demonstrated that candidate technologies, such as PC-RAM, FeRAM and STT-MRAM, can be considered as a replacement for DRAM. I also provided the overview of existing research projects and described how persistent memory can be integrated and supported by system software.

NV-Hypervisor As the third contribution, I developed a new model of persistence, named *hypervisor-based* model of persistence and the corresponding system support in the form of the NV-RAM-aware hypervisor.

In section 3.2, I demonstrated that legacy systems cannot profit from the introduction of NV-RAM because of limitations of existing models of persistence and a lack of necessary system support. NV-Hypervisor uses a name-based allocator with virtual memory support to provide transparent persistence for legacy and proprietary programs at the level of virtual machines. The evaluation of the hypervisor demonstrated that it can significantly decrease the performance recovery time of memory-consuming services.

4.1 Future work: Toward encrypted persistent systems

Persistence of main memory makes its content vulnerable to attacks based on physical access. In volatile systems, to perform a *cold-boot* attack, an attacker needs to make additional manipulations with memory modules: significantly decrease the temperature of the modules, remove them from the attacked hardware and place them into the attacker's platform. After that, the attacker can extract data. NV-RAM modules, in turn, do not require low temperatures to retain stored data in the absence of an external source of power, and thus, the content of those modules can be easily extracted. To prevent such attacks, persistent systems require memory encryption to protect the data stored inside, similarly as memory encryption protects volatile data [179]. This leads to the development of *encrypted persistent systems*.

A direct integration of existing technologies of memory encryption with persistent memory can protect the data, but it eliminates the persistence. Existing CPU extensions providing memory encryption, such as Intel SGX, AMD SME/SEV, do not expose the encryption keys to software: a CPU randomly generates these keys after power reset, and they can be neither read nor uploaded. Therefore, encrypted persistent memory loses encrypted data in the case of a power failure because the corresponding encryption key will be regenerated after the power recovery. In other words, encrypted persistent memory behaves like encrypted volatile memory.

The development of encrypted persistent systems requires both hardware and software improvements of existing solutions. Similar to ordinary persistent systems, several models of encrypted persistence could be possible. For example, the system-wide model of persistence can be used as a basis, and then, parts of the system should be selectively encrypted, in the same way as AMD SME enables encryption of memory pages. This approach will require additional support for system software since it will need to manage "volatile" memory regions. An alternative to this approach could be a hardware platform which provides read/write access to encryption keys, and the corresponding system software which ensures secure access to these keys.

In my future works I want to address these and other architectural challenges of persistent encrypted systems.

Bibliography

[1] M. Oros, "Analysts weigh in on persistent memory." [Online]. Available: https://www.snia.org/sites/default/files/PM-Summit/2018/presentations/14_PM_Summit_18_Analysts_Session_Oros_Final_Post_UPDATED_R2.pdf

[2] Everspin, "Mram into mainstream," 2016.

[3] Statista. (2018) Public cloud services: market size 2009-2021. Accessed 2018-06-08. [Online]. Available: https://www.statista.com/statistics/273818/global-revenue-generated-with-cloud-computing-since-2009/

[4] K. Panetta. (2017) Cloud Computing Enters its Second Decade. Accessed 2018-06-08. [Online]. Available: https://www.gartner.com/smarterwithgartner/cloud-computing-enters-its-second-decade/

[5] M. Ali, S. U. Khan, and A. V. Vasilakos, "Security in cloud computing: Opportunities and challenges," *Information sciences*, vol. 305, pp. 357–383, 2015.

[6] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of internet services and applications*, vol. 1, no. 1, pp. 7–18, 2010.

[7] CNET. (2014) Facebook sued for allegedly intercepting private messages. Accessed 2018-06-08. [Online]. Available: https://www.cnet.com/news/facebook-sued-for-allegedly-intercepting-private-messages/

[8] Forbes. (2014) eBay Suffers Massive Security Breach, All Users Must Change Their Passwords. Accessed 2018-06-08. [Online]. Available: https://www.forbes.com/sites/gordonkelly/2014/05/21/ebay-suffers-massive-security-breach-all-users-must-their-change-passwords/

[9] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," *ArXiv*, 2018.

[10] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *ArXiv e-prints*, Jan. 2018.

[11] H. Plattner and A. Zeier, *In-memory data management: technology and applications*. Springer Science & Business Media, 2012.

[12] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi, "Operating system support for NVM+ DRAM hybrid main memory," *Hot Topics in Operating Systems (HotOS)*, 2009.

[13] J. T. Pawlowski, "Hybrid memory cube (HMC)," in *Hot Chips 23 Symposium (HCS), 2011 IEEE*. IEEE, 2011, pp. 1–24.

[14] Ponemon Institute. (2016) Cost of Data Center Outages. Accessed 2018-06-08. [Online]. Available: https://www.ponemon.org/library/2016-cost-of-data-center-outages

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

[16] B. Böck and S. B. Austria, "Firewire-based physical security attacks on Windows 7, EFS and Bitlocker," *Secure Business Austria Research Lab*, 2009.

[17] G. E. Suh, C. W. O'Donnell, and S. Devadas, "AEGIS: A single-chip secure processor," *Information Security Technical Report*, vol. 10, no. 2, pp. 63–73, 2005.

[18] D. Kaplan, J. Powell, and T. Woller, "Amd memory encryption," Tech. Rep., apr 2016.

[19] 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 International Workshop on Hardware and Architectural Support for Security and Privacy*.

[20] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using Innovative Instructions to Create Trustworthy Software Solutions," in *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, p. 11.

[21] O. Weisse, V. Bertacco, and T. Austin, "Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 81–93.

[22] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: ExitLess OS Services for SGX Enclaves," in *EuroSys*, 2017, pp. 238–253.

[23] Intel, "Intel® Software Guard Extensions SDK for Linux* OS , Revision 2.0," https://01.org/intel-software-guard-extensions/documentation/intel-sgx-sdk-developer-reference, 2017.

[24] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. R. Pietzuch, and R. Kapitza, "SecureKeeper: Confidential ZooKeeper using Intel SGX." in *Middleware*, 2016, p. 14.

[25] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell *et al.*, "SCONE: Secure Linux Containers with Intel SGX." in *OSDI*, 2016, pp. 689–703.

[26] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, p. 8, 2015.

[27] C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX," in *2017 USENIX Annual Technical Conference*, 2017, pp. 645–658.

[28] M. Wuttig, "Phase-change materials: towards a universal memory?" *Nature materials*, vol. 4, no. 4, pp. 265–266, 2005.

[29] I. Cutress and B. Tallis, "Intel Launches Optane DIMMs Up To 512GB: Apache Pass Is Here!" *URL https://www.anandtech.com/show/12828/intel-launches-optane-dimms-up-to-512gb-apache-pass-is-here*, 2018, accessed 2018-11-22.

[30] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 2–13.

[31] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy, "Operating system implications of fast, cheap, non-volatile memory." in *HotOS*, vol. 13, 2011, pp. 2–2.

[32] R. F. Freitas and W. W. Wilcke, "Storage-class memory: The next storage system technology," *IBM Journal of Research and Development*, vol. 52, no. 4/5, p. 439, 2008.

[33] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy, "Overview of candidate device technologies for storage-class memory," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 449–464, 2008.

[34] M. K. Qureshi, M. M. Franceschini, A. Jagmohan, and L. A. Lastras, "Preset: improving performance of phase change memories by exploiting asymmetry in write times," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 380–391, 2012.

[35] A. Martens, R. Scholz, P. Lindow, N. Lehnfeld, M. A. Kastner, and R. Kapitza, "Dependable Non-Volatile Memory," in *Proceedings of the 11th ACM International Systems and Storage Conference*, ser. SYSTOR '18, 2018, pp. 1–12.

[36] A. C. Bomberger, N. Hardy, A. Peri, F. Charles, R. Landau, W. S. Frantz, J. S. Shapiro, and A. C. Hardy, "The KeyKOS nanokernel architecture," in *In Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, 1992.

[37] J. S. Shapiro, J. M. Smith, and D. J. Farber, *EROS: a fast capability system*. ACM, 1999, vol. 33, no. 5.

[38] J. S. Shapiro, E. Northup, M. S. Doerrie, S. Sridhar, N. H. Walfield, and M. Brinkmann, "Coyotos microkernel specification," *The EROS Group, LLC, 0.5 edition*, 2007.

[39] R. Welland, G. Seitz, L.-W. Wang, L. Dyer, T. Harrington, and D. Culbert, "The newton operating system," in *Compcon Spring'94, Digest of Papers*. IEEE, 1994, pp. 148–155.

[40] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1. ACM, 2011, pp. 91–104.

[41] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1. ACM, 2011, pp. 105–118.

[42] X. Li, K. Lu, X. Wang, and X. Zhou, "NV-process: A Fault-tolerance Process Model Based on Non-volatile Memory," pp. 1–1, 2012.

[43] D. Narayanan and O. Hodson, "Whole-system persistence," in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1. ACM, 2012, pp. 401–410.

[44] P. Mell, T. Grance *et al.*, "The NIST definition of cloud computing," 2011.

[45] T. Y. Woo, R. Bindignavle, S. Su, and S. S. Lam, "SNP: An Interface for Secure Network Programming." in *USENIX Summer*, 1994, pp. 45–58.

[46] K. Hickman and T. Elgamal, "The SSL protocol," 1995.

[47] T. Dierks and C. Allen, "RFC 2246: The TLS protocol version 1.0, January 1999," *Status: Proposed Standard*, 1996.

[48] J. Gotzfried and T. Muller, "ARMORED: cpu-bound encryption for android-driven ARM devices," in *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*. IEEE, 2013, pp. 161–168.

[49] B. Garmany and T. Müller, "PRIME: private RSA infrastructure for memory-less encryption," in *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 2013, pp. 149–158.

[50] L. Guan, J. Lin, B. Luo, and J. Jing, "Copker: Computing with Private Keys without RAM," in *NDSS*, 2014, pp. 23–26.

[51] L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang, "Protecting private keys against memory disclosure attacks using hardware transactional memory," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 3–19.

[52] V. Costan and S. Devadas, "Intel SGX Explained." *IACR Cryptology ePrint Archive*, p. 86, 2016.

[53] Intel, "Architecture instruction set extensions programming reference," 2014. [Online]. Available: https://software.intel.com/en-us/isa-extensions

[54] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen, "Intel® Software Guard Extensions: EPID Provisioning and Attestation Services," *White Paper*, vol. 1, pp. 1–10, 2016.

[55] S. Chakrabarti, R. Leslie-Hurd, M. Vij, F. McKeen, C. Rozas, D. Caspi, I. Alexandrovich, and I. Anati, "Intel® Software Guard Extensions (Intel® SGX) Architecture for Oversubscription of Secure Memory in a Virtualized Environment," in *Proceedings of the Hardware and Architectural Support for Security and Privacy*, 2017, p. 7.

[56] Intel. (2017) Intel software guard extensions for linux os. [Online]. Available: https://01.org/intel-softwareguard-extensions

[57] S. Shinde, D. Le Tien, S. Tople, and P. Saxena, "PANOPLY: Low-TCB Linux Applications With SGX Enclaves," in *Proc. of the Annual Network and Distributed System Security Symp.(NDSS)*, 2017.

[58] S. Arnautov, P. Felber, C. Fetzer, and B. Trach, "FFQ: A Fast Single-Producer/Multiple-Consumer Concurrent FIFO Queue," in *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 2017, pp. 907–916.

[59] S. Checkoway and H. Shacham, *Iago attacks: why the system call API is a bad untrusted RPC interface*. ACM, 2013, vol. 41, no. 1.

[60] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Security and Privacy (SP), 2015 IEEE Symposium on*, 2015, pp. 640–656.

[61] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *Proceedings of the 26th USENIX Security Symposium*, 2017.

[62] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza, "AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves," in *Proceedings of the 21st European Symposium on Research in Computer Security (ESORICS 2016)*, 2016, pp. 440–457.

[63] 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*. ACM, 2016, pp. 317–328.

[64] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 557–574.

[65] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using SGX to conceal cache attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 3–24.

[66] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "CacheZoom: How SGX amplifies the power of cache attacks," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2017, pp. 69–90.

[67] M. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: eradicating controlled-channel attacks against enclave programs," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*.

[68] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting privileged side-channel attacks in shielded execution with déjá vu," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 7–18.

[69] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[70] S. Chandra, V. Karande, Z. Lin, L. Khan, M. Kantarcioglu, and B. Thuraisingham, "Securing data analytics on SGX with randomization," in *European Symposium on Research in Computer Security*. Springer, 2017, pp. 352–369.

[71] S. Sasy, S. Gorbunov, and C. W. Fletcher, "ZeroTrace: Oblivious memory primitives from Intel SGX," in *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[72] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer, "SGXBOUNDS: Memory Safety for Shielded Execution," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 205–221.

[73] J. Seo, B. Lee, S. M. Kim, M. Shih, I. Shin, D. Han, and T. Kim, "SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.

[74] H. Tian, Q. Zhang, S. Yan, A. Rudnitsky, L. Shacham, R. Yariv, and N. Milshten, "Switchless Calls Made Practical in Intel SGX," in *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, 2018, pp. 22–27.

[75] Intel, "Intel® Software Guard Extensions SDK for Linux* OS , Revision 2.2," https://download.01.org/intel-sgx/linux-2.2/docs/Intel_SGX_Developer_Reference_Linux_2.2_Open_Source.pdf, 2018.

[76] M. Taassori, A. Shafiee, and R. Balasubramonian, "VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 665–678.

[77] J. Gu, Z. Hua, Y. Xia, H. Chen, B. Zang, H. Guan, and J. Li, "Secure live migration of SGX enclaves on untrusted cloud," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 225–236.

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

[79] N. Weichbrodt, P.-L. Aublin, and R. Kapitza, "Sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves," in *Proceedings of the 19th International Middleware Conference*, ser. Middleware '18, 2018, pp. 201–213.

[80] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, "A study of Linux file system evolution," *ACM Transactions on Storage (TOS)*, vol. 10, no. 1, p. 3, 2014.

[81] C. Hewitt, P. Bishop, and R. Steiger, "Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence," in *Advance Papers of the Conference*, vol. 3. Stanford Research Institute, 1973, p. 235.

[82] D. Charousset, R. Hiesgen, and T. C. Schmidt, "Caf – the C++ actor framework for scalable and resource-efficient applications," in *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control*. ACM, 2014, pp. 15–28.

[83] P. Haller and M. Odersky, "Scala Actors: Unifying thread-based and event-based programming," *Theor. Comput. Sci.*, vol. 410, no. 2-3, pp. 202–220, 2009.

[84] R. Rajwar and J. R. Goodman, "Speculative lock elision: Enabling highly concurrent multithreaded execution," in *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 2001, pp. 294–305.

[85] G. E. Collins, "A Method for Overlapping and Erasure of Lists," *Commun. ACM*, vol. 3, no. 12, pp. 655–657, 1960.

[86] R. Love, S. H. W. Are, A. C. Linus, and B. W. Begin, *Linux kernel development second edition*. Novell Press, 2005.

[87] A. Rubini and J. Corbet, *Linux device drivers*. " O'Reilly Media, Inc.", 2001.

[88] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza, "Rollback and Forking Detection for Trusted Execution Environments using Lightweight Collective Memory," in *Proceedings of the 47th International Conference on Dependable Systems and Networks*, ser. DSN'17, 2017, pp. 157–168.

[89] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "ROTE: Rollback protection for trusted execution," in *26th USENIX Security Symposium (USENIX Security 17)*, pp. 1289–1306.

[90] G. L. Peterson, "Myths about the mutual exclusion problem," *Information Processing Letters*, vol. 12, no. 3, pp. 115–116, 1981.

[91] C. Clifton, M. Kantarcioglu, J. Vaidya, X. Lin, and M. Y. Zhu, "Tools for Privacy Preserving Distributed Data Mining," *SIGKDD Explor. Newsl.*, vol. 4, no. 2, pp. 28–34, Dec. 2002.

[92] Y. Lindell and B. Pinkas, "Secure Multiparty Computation for Privacy-Preserving Data Mining," *IACR Cryptology ePrint Archive*, vol. 2008, p. 197, 2008.

[93] P. Saint-Andre, "Extensible messaging and presence protocol (XMPP): Core," 2011.

[94] P. Rösler, C. Mainka, and J. Schwenk, "More is Less: On the End-to-End Security of Group Chats in Signal, WhatsApp, and Threema," Cryptology ePrint Archive, Report 2017/713, 2017.

[95] libstrophe - an xmpp library for c. http://strophe.im/libstrophe/.

[96] *6th Gen Intel Core X-Series Processor Family Datasheet, Vol. 1*, jun 2017.

[97] G. Agha, I. A. Mason, S. Smith, and C. Talcott, "Towards a theory of actor computation," in *International Conference on Concurrency Theory*. Springer, 1992, pp. 565–579.

[98] L. V. Kale and S. Krishnan, "CHARM++: a portable concurrent object oriented system based on C++," in *ACM Sigplan Notices*, vol. 28, no. 10. ACM, 1993, pp. 91–108.

[99] D. G. Kafura and K. H. Lee, "ACT++: building a concurrent C++ with actors," 1989.

[100] J. Armstrong, "Erlang—a survey of the language and its industrial applications," in *Proc. INAP*, vol. 96, 1996.

[101] D. Charousset, T. C. Schmidt, R. Hiesgen, and M. Wählisch, "Native actors: a scalable software platform for distributed, heterogeneous environments," in *Proceedings of the 2013 workshop on Programming based on actors, agents, and decentralized control*. ACM, 2013, pp. 87–96.

[102] S. Srinivasan and A. Mycroft, "Kilim: Isolation-typed actors for java." in *ECOOP*, vol. 8. Springer, 2008, pp. 104–128.

[103] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy Data Analytics in the Cloud Using SGX," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 38–54.

[104] A. Havet, R. Pires, P. Felber, M. Pasin, R. Rouvoy, and V. Schiavoni, "SecureStreams: A Reactive Middleware Framework for Secure Data Stream Processing," in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. ACM, 2017, pp. 124–133.

[105] S. Brenner, T. Hundt, G. Mazzeo, and R. Kapitza, "Secure Cloud Micro Services using Intel SGX," in *Proceedings of the 17th International IFIP Conference on Distributed Applications and Interoperable Systems (DAIS '17)*, 2017.

[106] B. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov, "Iron: functional encryption using Intel SGX," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 765–782.

[107] R. Bahmani, M. Barbosa, F. Brasser, B. Portela, A.-R. Sadeghi, G. Scerri, and B. Warinschi, "Secure multiparty computation from sgx," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 477–497.

[108] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "Hydra: The kernel of a multiprocessor operating system," *Communications of the ACM*, vol. 17, no. 6, pp. 337–345, 1974.

[109] P. B. Hansen, "The evolution of operating systems," in *Classic operating systems*. Springer, 2001, pp. 1–34.

[110] H. Härtig, M. Roitzsch, C. Weinhold, and A. Lackorzynski, "Lateral Thinking for Trustworthy Apps," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 2017, pp. 1890–1899.

[111] "Redis," http://redis.io/.

[112] "Memcached," https://memcached.org.

[113] "Ignite," https://ignite.apache.org.

[114] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: protecting confidentiality with encrypted query processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*, 2011, pp. 85–100.

[115] S. D. Tetali, M. Lesani, R. Majumdar, and T. Millstein, "MrCrypt: Static analysis for secure cloud computations," *ACM Sigplan Notices*, vol. 48, no. 10, pp. 271–286, 2013.

[116] "MariaDB," http://mariadb.org/.

[117] "PostgreSQL," http://posgresql.org/.

[118] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy data analytics in the cloud using SGX," in *Security and Privacy (SP), 2015 IEEE Symposium on*, 2015, pp. 38–54.

[119] N. Balakrishnan, L. Carata, T. Bytheway, R. Sohan, and A. Hopper, "Non-repudiable disk I/O in untrusted kernels," in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, 2017, p. 24.

[120] M. Owens and G. Allen, *The Definitive Guide to SQLite*. Springer, 2010.

[121] P. Syverson, "A taxonomy of replay attacks [cryptographic protocols]," in *Computer Security Foundations Workshop VII, 1994. CSFW 7. Proceedings*. IEEE, 1994, pp. 187–191.

[122] Speedtest1 benchmarking suite. Accessed 2018-09-22. [Online]. Available: http://www.sqlite.org/src/finfo?name=test/speedtest1.c

[123] Council, Transaction Processing Performance, "TPC benchmark C (standard specification, revision 5.11), 2010," *URL: http://www.tpc.org/tpcc*.

[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, 2014.

[125] A. Arasu, S. Blanas, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, R. Ramamurthy, P. Upadhyaya, and R. Venkatesan, "Secure Database-as-a-service with Cipherbase," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 1033–1036.

[126] C. Mitchell, Y. Geng, and J. Li, "Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store," in *USENIX Annual Technical Conference*, 2013, pp. 103–114.

[127] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A holistic approach to fast in-memory key-value storage," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 429–444.

[128] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, 2014, pp. 295–306.

[129] C. Priebe, K. Vaswani, and M. Costa, "EnclaveDB: A Secure Database using SGX," in *EnclaveDB: A Secure Database using SGX*. IEEE, 2018, p. 0.

[130] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[131] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," *IEEE journal of solid-state circuits*, vol. 41, no. 3, pp. 712–727, 2006.

[132] S. A. Przybylski, *Cache and memory hierarchy design: a performance-directed approach*. Morgan Kaufmann, 1990.

[133] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, 2010.

[134] N. Setter, D. Damjanovic, L. Eng, G. Fox, S. Gevorgian, S. Hong, A. Kingon, H. Kohlstedt, N. Park, G. Stephenson *et al.*, "Ferroelectric thin films: Review of materials, properties, and applications," *Journal of Applied Physics*, vol. 100, no. 5, p. 051606, 2006.

[135] N. Nishimura, T. Hirai, A. Koganei, T. Ikeda, K. Okano, Y. Sekiguchi, and Y. Osada, "Magnetic tunnel junction device with perpendicular magnetization films for high-density magnetic random access memory," *Journal of applied physics*, vol. 91, no. 8, pp. 5246–5249, 2002.

[136] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell, "The Rio file cache: Surviving operating system crashes," in *ACM Sigplan Notices*, vol. 31, no. 9. ACM, 1996, pp. 74–83.

[137] AgigaTech., "AGIGRAM (TM) Non-Volatile System." http://www.agigatech.com/agigaram.php, 2012.

[138] "Viking Technology. ArxCis-NV (TM) Non-Volatile Memory Technology," http://www.vikingmodular.com/products/arxcis/arxcis.html, 2012.

[139] "DDR 3 / DDR4 NVDIMM Frequently Asked Questions. Viking Technology," http://www.vikingtechnology.com/nvdimm-faq, 2012.

[140] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *Micro, IEEE*, vol. 30, no. 1, pp. 143–143, 2010.

[141] M. E. Lines and A. M. Glass, *Principles and applications of ferroelectrics and related materials*. Oxford university press, 1977.

[142] Y. Huai, "Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects," *AAPPS Bulletin*, vol. 18, no. 6, pp. 33–40, 2008.

[143] I. Prejbeanu, M. Kerekes, R. C. Sousa, H. Sibuet, O. Redon, B. Dieny, and J. Nozières, “Thermally assisted MRAM,” *Journal of Physics: Condensed Matter*, vol. 19, no. 16, p. 165218, 2007.

[144] H. F. Hamann, C. H. Lam, M. L. Steen, and H.-S. P. Wong, “Multi-bit phase change memory cell and multi-bit phase change memory including the same, method of forming a multi-bit phase change memory, and method of programming a multi-bit phase change memory,” Feb. 3 2009, uS Patent 7,485,891.

[145] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, “Scalable high performance main memory system using phase-change memory technology,” in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 24–33.

[146] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, “Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling,” in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE, 2009, pp. 14–23.

[147] G. Dhiman, R. Ayoub, and T. Rosing, “PDRAM: a hybrid PRAM and DRAM main memory system,” in *Design Automation Conference, 2009. DAC’09. 46th ACM/IEEE*. IEEE, 2009, pp. 664–669.

[148] J. Zhao, O. Mutlu, and Y. Xie, “FIRM: Fair and high-performance memory control for persistent memory systems,” in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 153–165.

[149] R.-S. Liu, D.-Y. Shen, C.-L. Yang, S.-C. Yu, and C.-Y. M. Wang, “NVM Duet: Unified working memory and persistent store architecture,” *ACM Sigplan Notices*, vol. 49, no. 4, pp. 455–470, 2014.

[150] S. Schechter, G. H. Loh, K. Strauss, and D. Burger, “Use ECP, not ECC, for hard failures in resistive memories,” in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 141–152.

[151] N. H. Seong, D. H. Woo, V. Srinivasan, J. A. Rivers, and H.-H. S. Lee, “SAFER: Stuck-at-fault error recovery for memories,” in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2010, pp. 115–124.

[152] D. H. Yoon, N. Muralimanohar, J. Chang, P. Ranganathan, N. P. Jouppi, and M. Erez, “FREE-p: Protecting non-volatile memory against both hard and soft errors,” in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE, 2011, pp. 466–477.

[153] M. K. Qureshi, “Pay-As-You-Go: low-overhead hard-error correction for phase change memories,” in *Microarchitecture (MICRO), 2011 44th Annual IEEE/ACM International Symposium on*. IEEE, 2011, pp. 318–328.

[154] Direct Access for files. Accessed 2018-09-22. [Online]. Available: https://www.kernel.org/doc/Documentation/filesystems/dax.txt

[155] M. Wu and W. Zwaenepoel, “eNVy: a non-volatile, main memory storage system,” in *ACM SIGOPS Operating Systems Review*, vol. 28, no. 5. ACM, 1994, pp. 86–97.

[156] A.-I. Wang, P. L. Reiher, G. J. Popek, and G. H. Kuenning, “Conquest: Better Performance Through a Disk/Persistent-RAM Hybrid File System,” in *USENIX Annual Technical Conference, General Track*, 2002, pp. 15–28.

[157] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, “Better I/O through byte-addressable, persistent memory,” in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 133–146.

[158] D. Hitz, J. Lau, and M. A. Malcolm, “File System Design for an NFS File Server Appliance.” in *USENIX winter*, vol. 94, 1994.

[159] X. Wu and A. Reddy, “SCMFS: a file system for storage class memory,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 39.

[160] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, “System software for persistent memory,” in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 15.

[161] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: flexible file-system interfaces to storage-class memory," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 14.

[162] J. Xu and S. Swanson, "NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories." in *FAST*, 2016, pp. 323–338.

[163] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff, "NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 478–496.

[164] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *ACM SIGPLAN Notices*, vol. 49, no. 10. ACM, 2014, pp. 433–452.

[165] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Makalu: Fast recoverable allocation of non-volatile memory," in *ACM SIGPLAN Notices*, vol. 51, no. 10. ACM, 2016, pp. 677–694.

[166] A. Lindström, A. Dearle, R. Di Bona, S. Norris, J. Rosenberg, and F. Vaughan, "Persistence in the grasshopper kernel," *AUSTRALIAN COMPUTER SCIENCE COMMUNICATIONS*, vol. 17, pp. 329–338, 1995.

[167] A. Dearle, R. Di Bona, J. Farrow, F. Henskens, A. Lindström, J. Rosenberg, and F. Vaughan, "Grasshopper: An orthogonally persistent operating system," *Computing Systems*, vol. 7, no. 3, pp. 289–312, 1994.

[168] Ponemon Institute. (2011) Cost of Data Center Outages. Accessed 2018-06-08. [Online]. Available: https://www.ponemon.org/library/2011-cost-of-data-center-outages-sponsored-by-emerson-network-power

[169] ——. (2013) Cost of Data Center Outages. Accessed 2018-06-08. [Online]. Available: https://www.ponemon.org/library/2013-cost-of-data-center-outages

[170] J. S. Plank, M. Beck, G. Kingsley, and K. Li, *Libckpt: Transparent checkpointing under unix*. University of Tennessee], Computer Science Department, 1994.

[171] 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 USENIX Symposium on Networked Systems Design and Implementation*, 2008, pp. 161–174.

[172] G. Heiser, E. Le Sueur, A. Danis, A. Budzynowski, T.-l. Salomie, and G. Alonso, "RapiLog: reducing system complexity through verification," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 323–336.

[173] S. Kannan, A. Gavrilovska, and K. Schwan, "pVM: persistent virtual memory for efficient capacity scaling and object storage," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 13.

[174] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: closing the performance gap between systems with and without persistence support," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013, pp. 421–432.

[175] D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann, "Mini-ckpts: Surviving os failures in persistent memory," in *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 2016, p. 7.

[176] R. Ohmura, N. Yamasaki, and Y. Anzai, "Device state recovery in non-volatile main memory systems," in *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*. IEEE, 2003, pp. 16–21.

[177] Y. Tamura, "Kemari: Virtual machine synchronization for fault tolerance using domt," *Xen Summit*, 2008.

[178] M. R. Hines. (2013) RDMA Migration and RDMA Fault Tolerance for QEMU. [Online]. Available: http://www.linux-kvm.org/images/0/09/Kvm-forum-2013-rdma.pdf

[179] P. Simmons, "Security through amnesia: a software-based solution to the cold boot attack on disk encryption," in *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011, pp. 73–82.