Master's Thesis

Hardening Applications with Intel SGX

Anwendungen härten mit Intel SGX

Fredrik Teschke

Supervisors
Prof. Dr. Andreas Polze, Max Plauth
Operating Systems and Middleware Group

Hasso-Plattner-Institut at the University of Potsdam

July 17, 2017

Abstract

The release of Intel SGX revived the interest in trusted computing across industry and academia. Hardware is available, but usage patterns and applications are mostly missing. This thesis evaluates trusted computing from the viewpoint of a software engineer. Hardening strategies are identified in related work and applied in two case studies. The case studies show how SGX can be used in practice. A small helper library is developed for rapid prototyping. Other trusted computing solutions are compared to SGX and SGX is critically evaluated based on current research.

Zusammenfassung

Die Veröffentlichung von Intel SGX hat das Interesse an Trusted Computing in Akademia und Industrie wieder erweckt. Obwohl die Hardware verfügbar ist, sind Verwendungsmuster and Anwendungen noch Mangelware. Diese Arbeit evaluiert Trusted Computing aus der Sicht eines Software Ingenieurs. In verwandten Arbeiten werden Strategien zum Härten von Anwendungen identifiziert und in zwei Fallstudien angewandt. Diese Fallstudien zeigen wie SGX praktisch genutzt werden kann. Dabei wird eine kleine Hilfs-Bibliothek entwickelt. Andere Trusted Computing Lösungen werden mit SGX verglichen und SGX wird mit Hilfe aktueller Forschungsergebnisse kritisch bewertet.

Contents

1. Introduction **1**

2. Background **3**
2.1. Cryptography 5
2.2. Trusted Computing 8

3. Trusted Computing Solutions **11**
3.1. Classification 11
3.2. Commercial 11
3.3. Research 16
3.3.1. Module Isolation 16
3.3.2. Application Isolation 19
3.3.3. Operating System Isolation 22
3.4. Comparison 23

4. Intel SGX **27**
4.1. Overview 27
4.2. Enclave Development 30
4.3. Performance 32
4.4. Known Criticism 33
4.5. Applications 36
4.6. Conclusion 37

5. Intel SGX Helper Library **39**

6. Related Work **45**
6.1. Hardened Databases 45
6.2. Hardening Applications with Intel SGX 48

7. KISSDB Case Study **51**
7.1. Design 51
7.2. Implementation 53
7.3. Open Issues 56
7.4. Conclusion 57

8. SQLite Case Study **59**
8.1. Analysis 60

8.2. Concepts 60
8.3. SQLite in the Intel SGX SDK 65
8.4. Conclusion 65

9. Conclusion **67**

A. KISSDB Database Files **69**

B. SQLite Call Graphs **71**

References **73**

1. Introduction

Cloud computing has proven itself as a viable and popular business model. This makes data security an increasingly hot topic. Encryption as a way of securely transporting data is an age-old and proven concept. By comparison, techniques for secure data *processing* are still in their infancy.

For some decades there have been niche solutions in this field called trusted computing. They did not gain the traction and publicity they may have deserved. Among such solutions are Trusted Platform Modules and ARM's TrustZone security extensions. Now Intel has joined the game and has been shipping its Security Guard Extensions (SGX) with many of its new CPUs since end of 2015. The wide-spread availability of trusted computing hardware is foreseeable. There is a growing demand for trustworthy applications in digital rights management and cloud computing. This means the game might soon begin to change, shifting trusted computing back into focus.

From a technological standpoint, trusted computing is fascinating. It combines the fields of cryptography, operating systems and hardware design. However, from an ethical standpoint, trusted computing is a double-edged sword as Figure 1.1 pointedly makes clear.

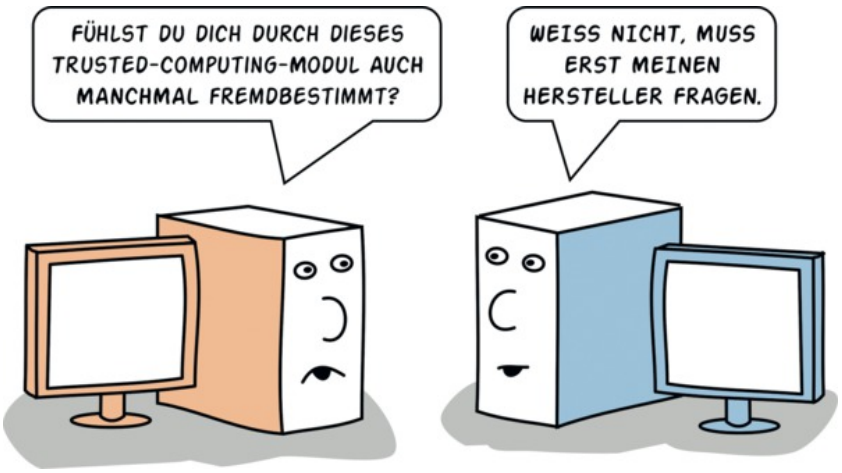


Figure 1.1.: Trusted computing cartoon. Left computer: "Do you also sometimes feel remotely controlled by this trusted computing module?". Right computer: "I don't know, let me ask my manufacturer." Reprinted from [52].

Intel SGX, and with it the field of trusted computing in general, still has to pick up traction. Yet the technology is ready for being used and evaluated today. A variety of research is happening around SGX, and innovative use cases are popping up.

This thesis looks at trusted computing from a *software engineer's perspective*. The goal of this thesis is to show how a developer can harden his applications today, using the technology that is available. The thesis is mainly made up of literature work. Techniques for secure remote computation are described, among them trusted computing (chapter 2). A wide variety of trusted computing solutions is then surveyed and systematically compared (chapter 3). Both commercial solutions and research work are included.

Intel SGX is chosen as the prime candidate for a more detailed evaluation (chapter 4). The SDK provided by Intel is presented along with a small helper library that was developed as part of this thesis (chapter 5). Architectural design patterns for hardening applications are identified in the related work (chapter 6). Two case studies show how database software – representative for the class of hosted applications – can be hardened using Intel SGX (chapter 7, chapter 8). These case studies use the patterns and techniques found in related work.

While SGX is an exciting technology that is in many regards better than previous solutions, it is far from perfect. Criticism and security issues are also presented.

The source code for this thesis – including all text, images and code snippets – is available at `https://github.com/ftes/sgx-thesis`.

2. Background

Nowadays, data is oftentimes not stored – and applications are not executed – locally (on premise) any more. Rather, these tasks are outsourced to hosted, remote infrastructure. In addition, computer technology is becoming increasingly pervasive in our lives. More data is stored on and processed by computers, making them ever more valuable targets.

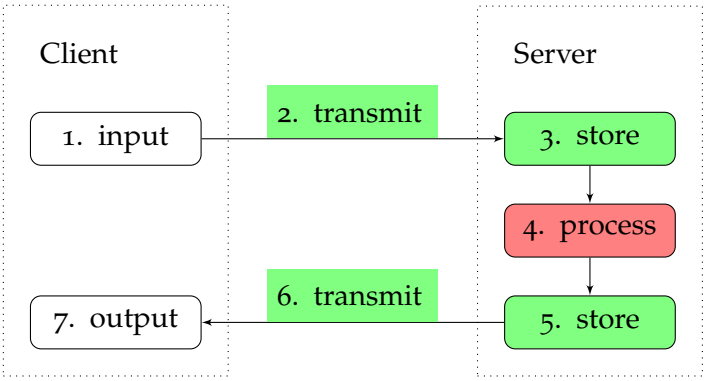


Figure 2.1.: Focus of this thesis within the data life cycle. Transmitting and storing data can be secured using encryption (green). The applications examined in this thesis deal with processing data (red). This is an area of active research.

The state of the art is to protect sensitive data by *encrypting* it while it is at rest or being transmitted. This is shown in Figure 2.1. Protecting the processing stage is an active field of research called *secure remote computation*.

Figure 2.2 gives an abstract overview of the entities and steps involved in secure remote computation. For the sake of this thesis, the most interesting part of the picture is the implementation of the container.

Arasu et al. categorise the approaches for constructing such a container that can protect code and data on a remote computer:[3, p. 19]

Compute on encrypted data The data remains encrypted during processing. Thus the results are also encrypted. In this case the cryptographic scheme is the container. No information about the plain text should be leaked. section 2.1 explains which encryption schemes support this.

Decrypt and process data in a secure location Such a location could be a local machine, disconnected from the internet, or a remote trusted hardware component such as a secure co-processor. Whether or not a location is deemed secure is a subjective decision. This variant of implementing the container is called *trusted computing*.

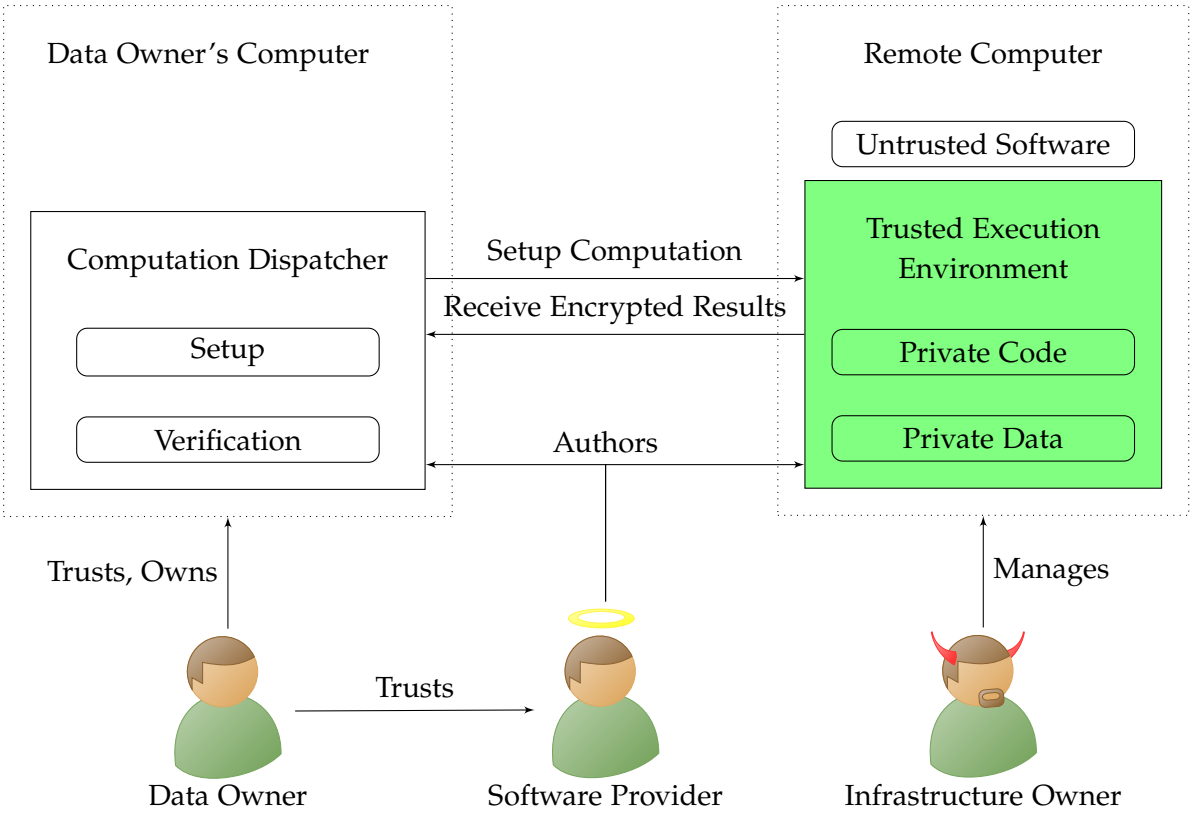


Figure 2.2.: Secure remote computation. The data owner trusts the software provider but not the infrastructure owner. The code and data within the trusted execution environment (green) must be protected. There are different options for implementing this protection. Reprinted from [15].

The remainder of this chapter explains the fundamental concepts of both these approaches.

Implementations of the first approach are presented later on in section 6.1. Implementations of trusted computing are described and compared in chapter 3. Intel SGX – a particular commercial solution for trusted computing – is described in more detail in chapter 4.

2.1. Cryptography

There are several different ways in which cryptographic principles can be used to implement the concept of a secure container.

Multi party computing Several parties jointly compute a function to which every party provides some input. The input of each party is not revealed to any of the other parties. One early implementation is Yao's garbled circuits.[27] For secure remote computation, we could assume two parties, where only the data owner provides an input and only the infrastructure owner executes the function. However, the function output is in plain text which is not desirable for secure remote computation in general.

Verifiable computing This is a first step in the direction of secure remote computing. It ensures the integrity but not the confidentiality of the computation (similar to a cryptographic signature).[23, 36, 61]

Homomorphic Encryption Such encryption schemes define calculation operations on encrypted data. The operands and result of these calculations remain encrypted so they could be performed by an untrusted third party. Figure 2.3 explains the principle of homomorphic encryption with an example. While partially homomorphic schemes define only one operation (e.g. either addition or multiplication), fully homomorphic schemes define both.

Gentry et al. successfully constructed the first fully homomorphic scheme in 2009.[24] Figure 2.4 shows the relationship between different encryption schemes and the operations they support. These schemes are revisited in section 6.1, which also shows how they can be practically put to use.

State of the art fully homomorphic schemes still suffer from an intractably high overhead. Partially homomorphic schemes on the other hand have already been applied to databases.[8, 3]

Encryption schemes in themselves also do not help verify what computation took place. Combining encryption with verifiable computation approaches or software attestation may provide a solution.

Encrypted CPU Given a (fully) homomorphic encryption scheme it is possible to execute entire encrypted programs. This is possible in a fully oblivious fashion where both the instruction flow and memory access (code and data) remain hidden. Both obliviousness and the current fully homomorphic encryption schemes incur such large performance penalties that they are not yet practically useful for more complex programs.

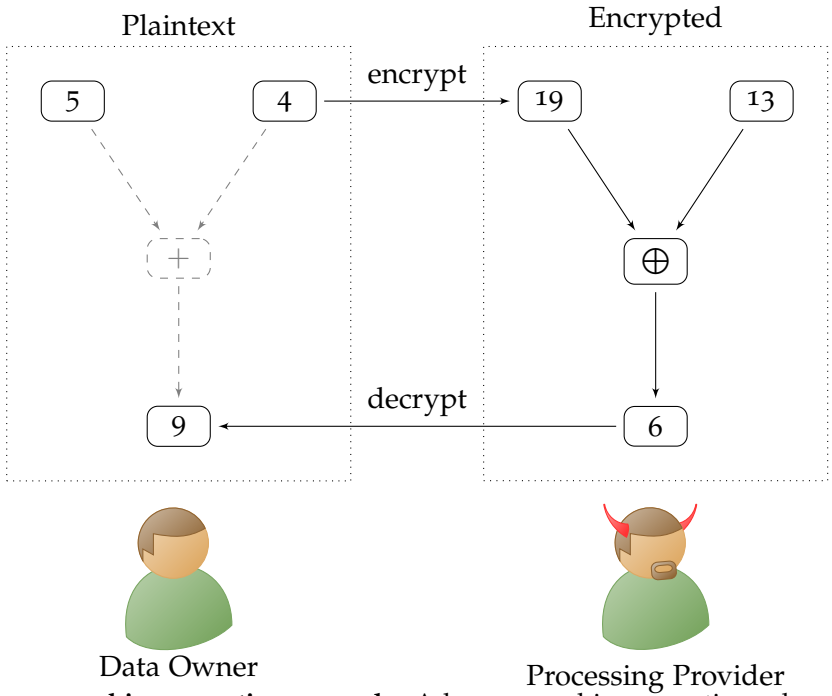


Figure 2.3.: Homomorphic encryption example. A homomorphic encryption scheme defines operations on encrypted data. The decrypted result of the encrypted addition ($\oplus$) gives the same result as performing a plain text addition ($+$). Using this scheme an untrusted *processing provider* can perform calculations without learning anything about the plain text.

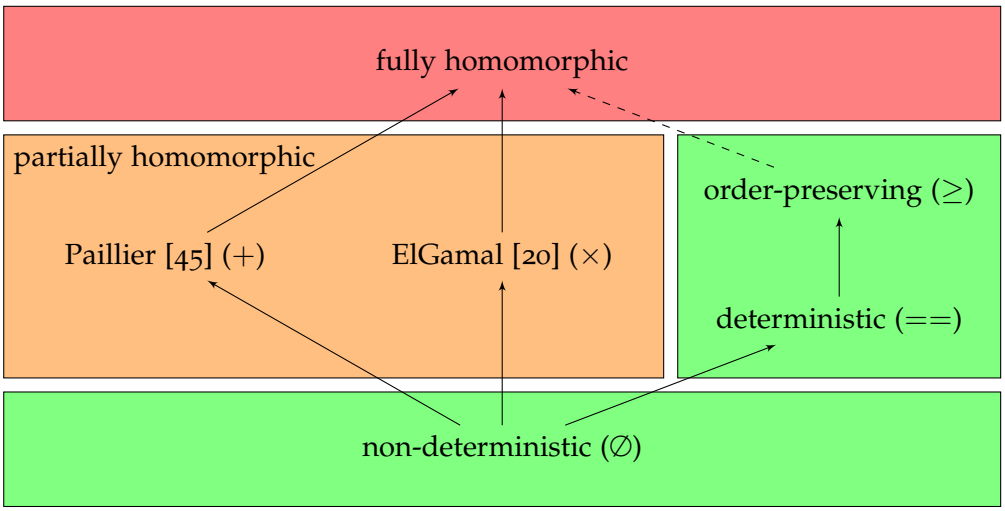


Figure 2.4.: Encryption schemes and their relationships. The shading indicates computational efficiency (red: impractical, orange: expensive, green: practical). Arrows indicate subsumption of functionality. Fully homomorphic schemes for example provide both $+$ and $\times$ operations (and by extension – e.g. an encrypted CPU – also comparison operations). Reprinted from [3].

The encrypted CPU works very much the same as a regular CPU, as can be seen in Figure 2.5. A regular CPU operates on bits using logical gates. The encrypted CPU operates on encrypted bits. Each bit is represented by cipher value of n bits size so that there are 2^{n-1} possible representations for 0 and 1 respectively. Logical gates are emulated using the fully homomorphic operations on these cipher values (addition and multiplication).[11]

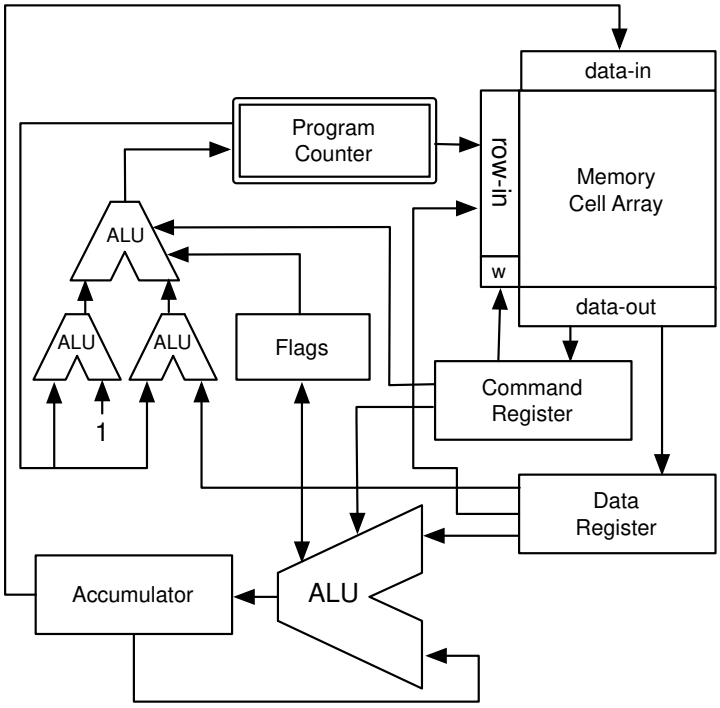


Figure 2.5.: Encrypted CPU schematic. This is a classic von-Neumann architecture where the memory holds both the instructions and data. Bits are represented as encrypted numbers. Reprinted from [10].

Both code and data reside in the encrypted memory. This means that code and data remain secret at all times.

The circuit evaluation provides *obliviousness* as the entire circuit must always be solved. For example, on memory accesses each cell is reassigned – either with its new value on a write, or an equivalent representation of its old bit value. This obliviousness is an important security factor and a performance pitfall at the same time. Memory access patterns and program flow are kept secret. This however also means that access times grow with the memory size. The authors state that "compact programs and data are the key to tolerable runtimes".[9]

A hardware implementation of the encrypted CPU has not yet been attempted due to various challenges such as super-wide buses and the recrypt procedure necessitated by the encryption scheme. A software implementation highlights the performance problems. Without encryption, a CPU cycle is simulated in $3ms$. With encryption this

value increases to 166s. [1] Both values are obtained for 256 memory rows – at 13bit per row this gives roughly 0.4kB of memory.[2][9]

Fully homomorphic encryption adds another tool to a cryptographer's toolbox: The ability to compute on encrypted data. The concept of an encrypted CPU builds on top of this primitive. It shows how a encrypted program with branching can be executed on encrypted data.

The performance of fully homomorphic schemes is still far from being practically applicable. Through the oblivious full-circuit evaluation of the encrypted CPU this issue is amplified. However, improvements are possible on several avenues: New, more efficient fully homomorphic schemes may be devised. The existing schemes can be optimised both in their algorithm and in their implementation (e.g. parallelised). Hardware implementation of the encryption, and especially the encrypted CPU also has large potential benefits.

Yet even a sufficiently efficient encrypted CPU could not solve secure remote computation once and for all. Firstly the computation is restricted to a single client. Without decrypting the results (in a trusted location) no communication and interaction between clients is possible. Secondly the problem of attestation is not solved by this approach.

2.2. Trusted Computing

This section defines terms important for trusted computing. These are most relevant for chapter 3.

Root of trust is the sole element on which trust in a platform hinges. If the root of trust is compromised, the whole platform is compromised.[21] For example, the CPU in a trusted computing setup could be the root of trust that is expected to function correctly.

Trusted computing is a form of secure remote computation that uses trusted hardware as the root of trust.[21] Figure 2.6 shows the involved components and trust relationships.

Trusted Execution Environment (TEE) protects its assets (such as code and data) from attacks. It usually exists alongside the standard Rich Execution Environment (REE).[25] The TEE is at the very heart of a trusted computing implementation as shown in Figure 2.6. This section describes different TEE implementations.

Trusted computing base (TCB) is best described by the *Orange Book*: The TCB "contains all of the elements of the system responsible for supporting the security policy".[18] This includes the root of trust, the application itself, and all intermediate software levels that have to be trusted. Anything outside of the TCB does not have to be trusted. The TCB should be as small and simple as possible for the sake of security.[18] Depending on

[1] This is the measurement for the highest value of the security parameter *lambda*. Unfortunately, neither Brenner et al. nor Smart et al.[55] give further details on how the security parameter relate to a comparable security level. The BSI advises a security level of 120bit from the year 2022 onwards: `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf`.

[2] A memory word contains 8bit of data and a 5bit command.[11]. This design decision reduces the number of costly memory access cycles.

the trusted computing solution the TCB may contain the operating system and/or the hypervisor.

Software Attestation is a two-part process. First a loaded piece of software is measured to ensure that the system is in a well-defined state. Secondly, this measurement is cryptographically signed and transmitted. This protocol can be enriched to include a key exchange. This makes it possible to securely communicate with the attested code. The process is described well in [15].

Data Sealing is a process of storing data so that it can only be accessed by a component in a certain state. For example, bank account credentials could be sealed so that they can only be read by a certain operating system at a certain patch-level.[21]

Technically, this is usually achieved through key derivation. The root of trust in a system may have a secret key. From this key, with the measurement result of software attestation, a state-specific data sealing key is derived. The data is then encrypted with this key.

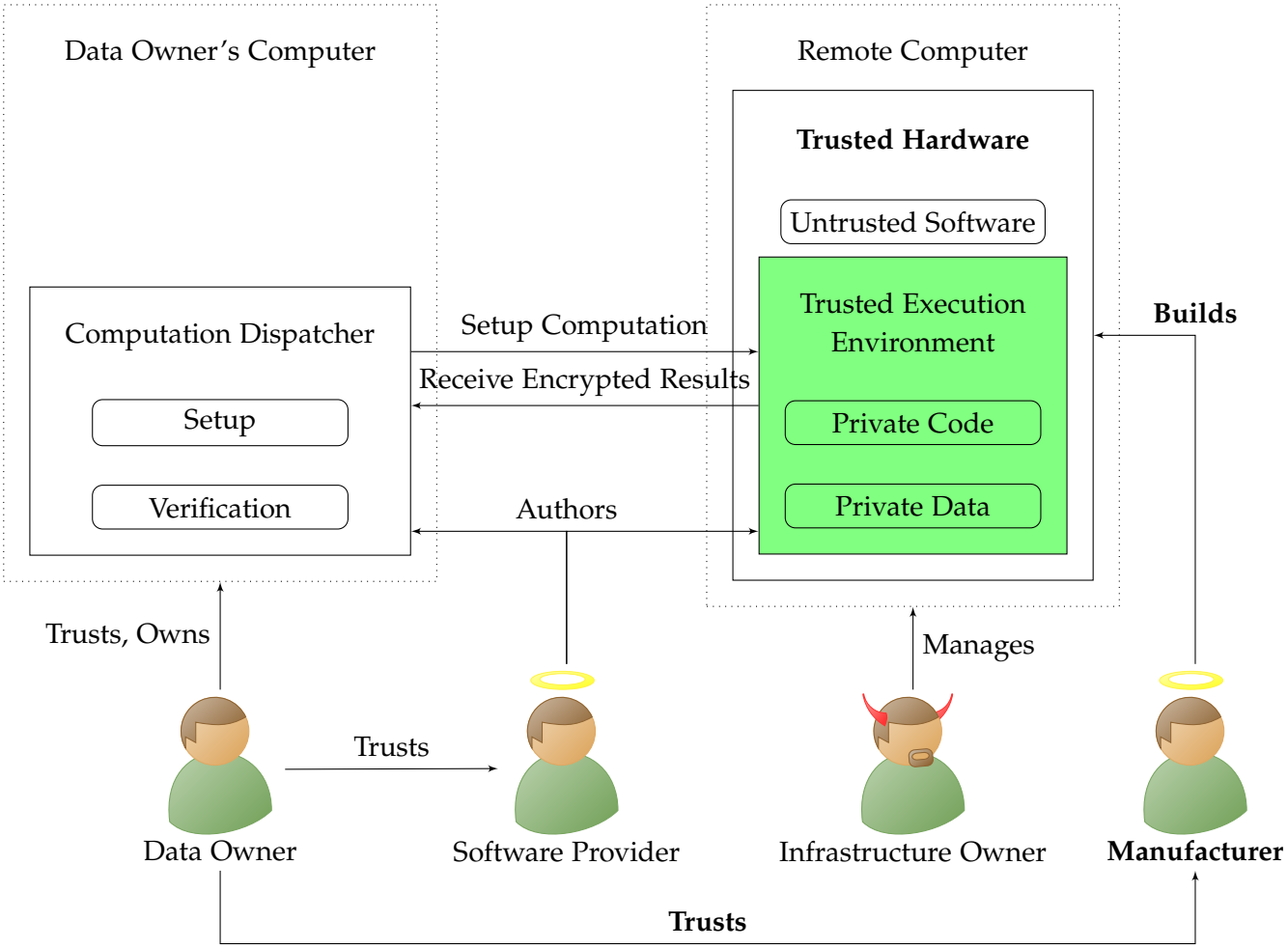


Figure 2.6.: Trusted computing. The trusted execution environment is protected by trusted hardware. This introduces an additional trust relationship. Additional nodes (compared to Figure 2.2) are in bold font. Reprinted from [15].

3. Trusted Computing Solutions

As explained, trusted computing is a variant of secure remote computing built on trusted hardware. This chapter first defines metrics for classifying trusted computing implementations. Commercially available solutions and solutions from research are then described qualitatively. Finally, a more quantitative comparison is given in form of a table. It uses the defined metrics as the main criteria.

3.1. Classification

The following dimensions are used to classify the solutions presented in the remainder of this section:

Hardware implementation (if present). Figure 3.1 shows a variety of approaches ranging from external to on-chip solutions as defined by the GlobalPlatform alliance.[25] Using hardware virtualisation techniques is a fourth option used in some solutions.

Isolation level at which the TEE protects the components. Figure 3.2 shows the five predominant isolation levels. These levels can be observed repeatedly when evaluating the trusted computing implementations presented in this thesis.

3.2. Commercial

The following list of commercial trusted computing solutions gives a good overview of how the field has evolved in the past 15 years. The list is not exhaustive. Instead, the chosen solutions represent noteworthy archetypes.[3] For a more extensive list, see [19].

2002: Trusted Platform Module (TPM)[4] is a separate component in a computer system that can be used for various cryptographic and attestation tasks.[59] It can be classified as a external secure element (Figure 3.1) that can – with different means – provide a variety of isolation levels (Figure 3.2). The TPM must maintain a separate state which cannot be tampered with. For this reason, TPMs are usually dedicated hardware chips.[5]

[3] The most noticeable omission from this list are all kinds of cryptographic co-processors that aim to provide significant computational resources apart from the main CPUs. Any operation in excess of cryptographic primitives such as key generation and digital signatures is considered significant.

[4] TPM hardware first became available for the revision 1.2 of the TPM specification. This was published in 2003: `https://trustedcomputinggroup.org/wp-content/uploads/tpmwg-mainrev62_Part1_Design_Principles.pdf`. Later, in 2009, the TPM specification was ISO standardised: `https://www.iso.org/standard/50970.html`

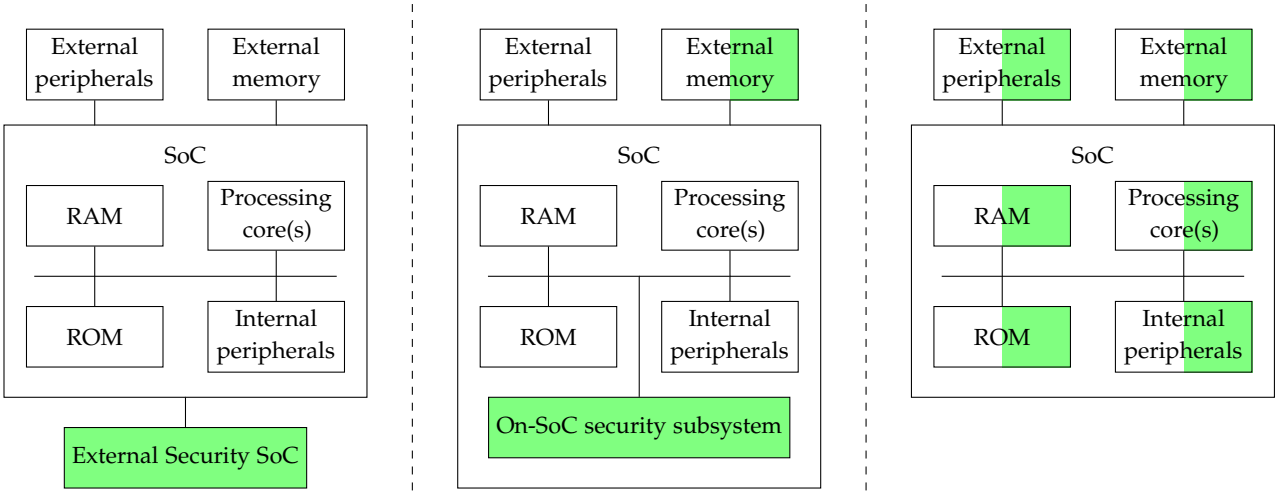


Figure 3.1.: Implementation alternatives for protecting a Trusted Execution Environment (TEE) as defined by the GlobalPlatform alliance. The logic necessary to protect the TEE lives in nodes shaded green. It can either reside outside of the System on a Chip (SoC) as in *a)*, or as a part of the regular SoC components as in *c)*. Reprinted from [25].

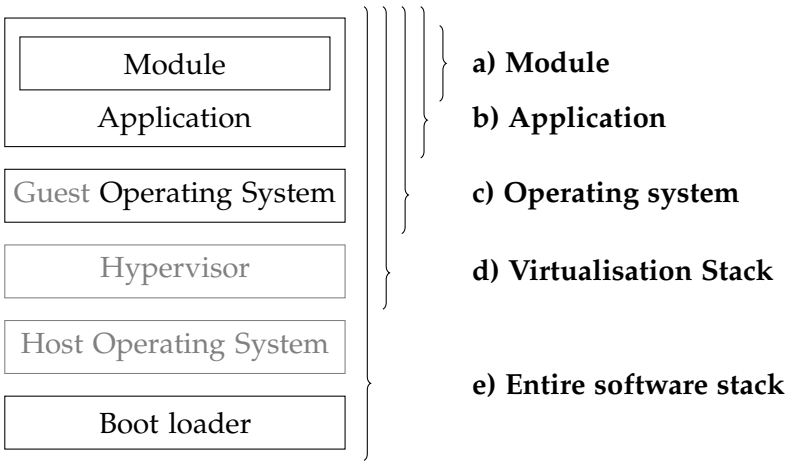


Figure 3.2.: Possible levels of isolation a Trusted Execution Environment (TEE) can provide. *a) – e)* represent the five predominant levels in the evaluated trusted computing solutions. Virtualisation is not employed by all solutions, therefor the *host operating system* and *hypervisor* are printed in grey.

A TPM has an embedded secret key used to sign its outputs, e.g. when supplying system measurements. This secret key is certified by the manufacturer to establish its authenticity.[58]

TPMs can be used to measure the state of the entire system. This can be done in a *static* fashion, starting from the boot loader, as shown in Figure 3.3. The TPM can also provide a *dynamic* measurement. This is done when software such as a hypervisor is elevated into a super-privileged virtual machine management (VMM) mode. Performing a dynamic measurement requires CPU support.[6] See section 3.3 for details on how TPMs can also be used to provide isolation for components on levels smaller than the virtualisation stack layer (*a-b* in Figure 3.2).

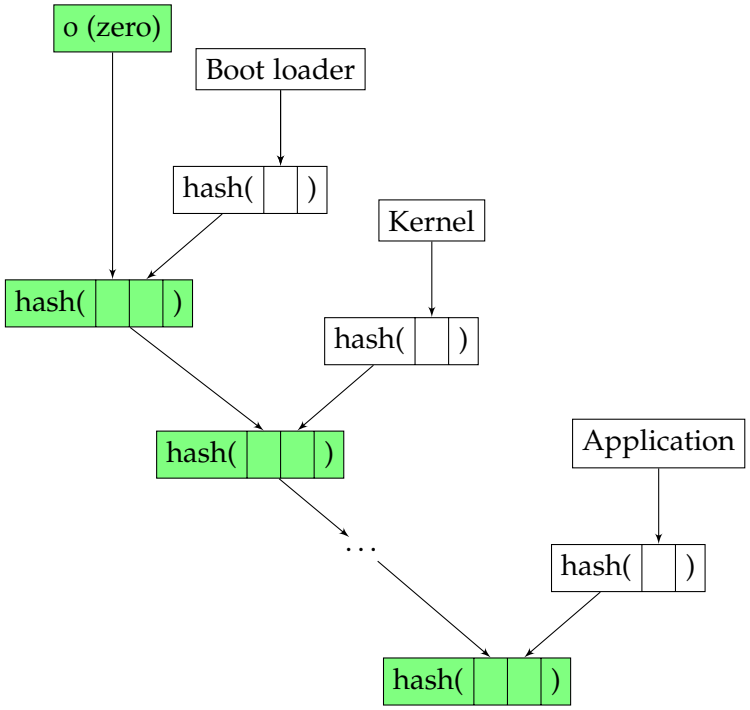


Figure 3.3.: Static system state measurement using a Trusted Platform Module (TPM). The TPM stores the measurement in a register (register values are shaded green). At reboot, the measurement register is reset to zero. Then "the software at every boot stage hashes the next boot stage".[15] This hash is sent to the TPM, which updates the measurement register by hashing both the old register value and new measurement. Reprinted from [15].

TPMs are not ideally suited for securing individual applications:

- TPMs do not isolate processes. Apart from trusted cryptographic functions they only provide a measurement of a software state. Isolation must be implemented in software. This is susceptible to privilege escalation. Also, this is an additional development and/or maintenance overhead.

[5] [49] describes a software TPM implementation using TrustZone. The TPM state is protected in the secure world.
[6] The technologies of the two major vendors are Intel TXT and AMD SVM.

- System components that must be trusted include the system bus and main memory.[35] A TPM can therefore not protect secrets from any party that has hardware access, and might e.g. read main memory.

2003: ARM TrustZone[7] is an optional extension to the ARM CPU specification. It can be classified as a processor secure environment (Figure 3.1) that provides isolation at the application level (Figure 3.2). A TrustZone-capable system can be described as having a split personality. It runs in either the normal world or the secure world, indicated by an extra bit on the system bus.[4] Other hardware components use this bit to implement access restrictions. For example the memory management unit does not allow access to pages that belong to the secure world while running in the normal world. A special instruction, the *secure monitor call*, lets the system switch worlds by executing the monitor code which was defined during system start up.

The distinction between normal and secure world is orthogonal to the regular privilege levels (user and kernel mode) as shown in Figure 3.4.

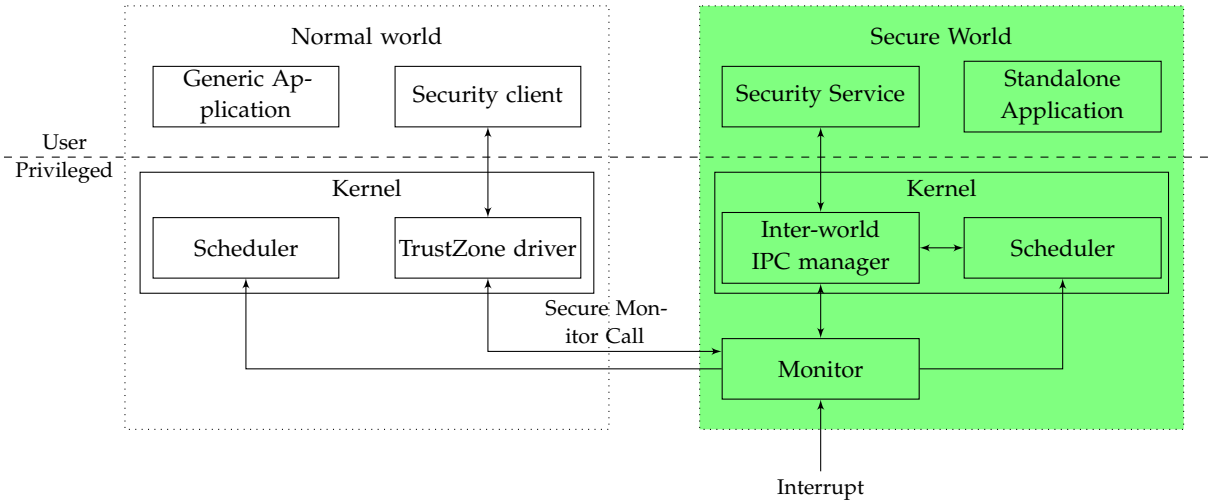


Figure 3.4.: Example secure world implementation using ARM TrustZone technology. The system boots in secure mode and a monitor is registered which acts as the interface between secure and normal world. The secure word has its own kernel which must handle process isolation. Applications in the normal world can indirectly access services in the secure world through a *secure monitor call*. Reprinted as a simplified version from [4].

The figure also shows that interrupts are first handled by the monitor. Devices can thus be mapped to either of the two worlds, or both. If a device, e.g. a keyboard, is mapped to the secure world it is possible to provide trusted input that cannot be tampered with by the normal world. If a device is mapped to both worlds (such as main memory) then the device controller must enforce the access restrictions (e.g. by keeping track of which world a memory page is assigned to via the page table).

[7] `https://www.arm.com/about/newsroom/3791.php`

TrustZone is a very flexible hardware concept. In its documentation, ARM proposes to implement two worlds with separate kernels. Samsung Knox on Android phones is a good example. Knox provides attestation capabilities and sets up an isolated workspace environment, which is completely separated from the regular environment.[50] As the hardware imposes no limits on how it is used, it is also possible to implement deviating concepts such as a firmware TPM.[49]

Attestation is not part of the TrustZone specification. However , approaches such as the firmware TPM show that this concept is easy to implement using TrustZone. The hardware root of trust is present. All that is needed in addition is a secret key only accessible by the secure world. TrustZone is – by itself – not strictly a trusted computing solution as a remote party cannot verify the state of the secure world. It is still included in this list because it can serve as a hardware basis to implement fully-fledged trusted computing solutions.

While TrustZone is flexible, it is not ideally suited for securing applications in a general fashion due to the following reasons:

- TrustZone isolates worlds, but not processes within the secure world. All applications that should be protected live together in the secure world. It is solely the responsibility of the Kernel to isolate the processes in the secure world. The data in the secure world is thus susceptible to be compromised via privilege escalation of the secure kernel.
- To isolate applications on TrustZone hardware, a monitor and secure kernel are needed. This is additional development overhead (or at least maintenance overhead[8]).
- The TCB is far larger than the security critical parts of the application that should be hardened. It includes the boot loader, monitor, secure kernel and all other applications running in the secure world.

2015: Intel Software Guard Extensions (SGX)[9] is an instruction set extension with which protected memory regions, called enclaves, can be set up. An enclave is a TEE for a single software module. It can be classified as a processor secure environment (Figure 3.1) that provides isolation at the module level (Figure 3.2). It is orthogonal to existing protection mechanisms such as virtual memory or privilege levels. Enclaves are protected from any external access not allowed by their interface definition, be it by the operating system or an administrator with hardware access.[42] Like a TPM, an SGX-enabled CPU has an embedded secret key so it can provide signed measurements of an enclave's state to third parties.[34] SGX is explained in more detail in chapter 4.

SGX is well-suited to secure applications:

- SGX isolates at the module level. The TCB consists of only the module code.

[8] https://github.com/ARM-software/arm-trusted-firmware
[9] https://software.intel.com/en-us/sgx

- No hardware apart from the CPU must be trusted. Memory is encrypted when stored in RAM.

2016: Windows Isolated User Mode (IUM)[10] is a secure execution mode similar to the secure world in TrustZone. It uses virtualisation (not shown in Figure 3.1) and provides isolation at the application level (Figure 3.2). The kernel and processes in secure mode are separated from normal mode by the Hyper-V hypervisor.[11] Unlike TrustZone and the other technologies in this list, IUM is implemented in software – not considering CPU virtualisation support. IUM is used to secure credentials in the Windows Credential Guard.[12].

IUM has limited potential for securing applications:

- The TCB size is large. It includes the hypervisor, secure kernel and application.
- Data in the isolated mode can be compromised via privilege escalation of the secure kernel.
- Microsoft has not yet published any information on how to develop applications for IUM. It seems that for now it is used for internal Windows functionality such as Credential Guard only.
- As a software-only solution, no hardware root of trust is present. Windows IUM does not provide attestation. Strictly speaking it does not match the definition of trusted computing used in this thesis. IUM is still listed, as it is comparable to many of the solutions from research.

3.3. Research

Trusted computing solutions from the research community are now introduced in detail. Where possible, similarities to the commercial solutions are pointed out. The solutions are grouped by isolation level (Figure 3.2). The order of the following solutions is the same as in Table 3.1, which gives a high-level comparison.

3.3.1. Module Isolation

Sanctum[16] Sanctum is comparable to Intel SGX in both implementation and features. As the authors themselves state, it "draws heavy inspiration" from SGX. It was designed by Costan and Devadas, who also reverse-engineered and documented many details of SGX.[15] Sanctum tries to improve on SGX. It protects against software attacks that analyse a program's memory access patterns.

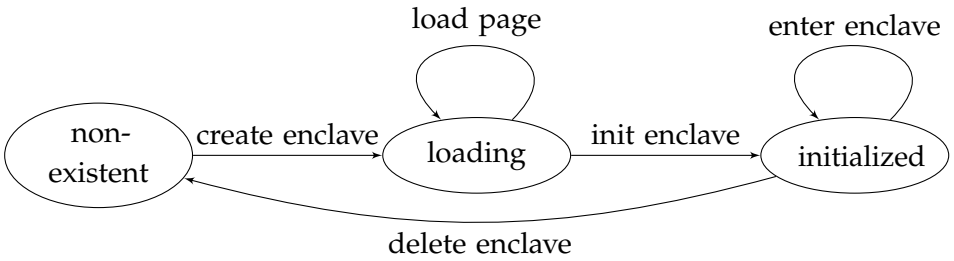
The implementation is less invasive than SGX, as it only "adds hardware at the interfaces between building blocks" instead of modifying them directly. Sanctum isolates enclaves

[10] https://msdn.microsoft.com/en-us/library/windows/desktop/mt809132(v=vs.85).aspx
[11] https://channel9.msdn.com/Blogs/Seth-Juarez/Isolated-User-Mode-in-Windows-10-with-Dave-Probert
[12] https://docs.microsoft.com/en-us/windows/access-protection/credential-guard/credential-guard

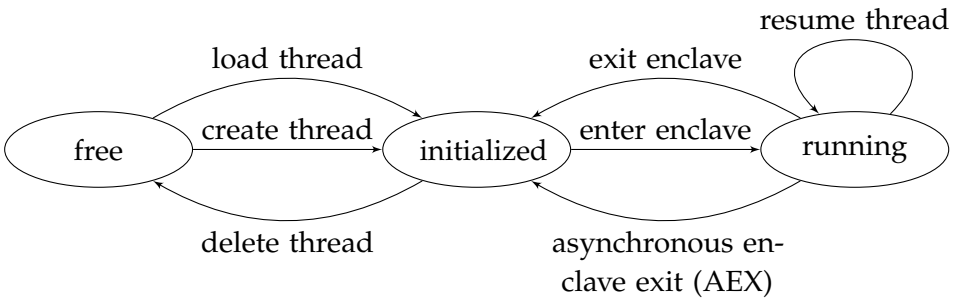
by virtually partitioning the DRAM into "regions that use disjoint Last Level Cache (LLC) sets." The page walker then enforces the access rules as known from SGX.

The hardware additions are complemented by a security monitor. It is small enough to be formally verified. The monitor is responsible for handling "DRAM region allocation and enclave management" and protects sensitive registers.[16]

Without going into too much detail, Figure 3.5 shows how similar Sanctum's enclave and thread management are to SGX.



a) Enclave state diagram



b) Enclave thread state diagram

Figure 3.5.: State diagrams for enclave and thread state in Sanctum. The states and transitions are very similar to those in SGX since Sanctum's design was largely inspired by SGX. Reprinted from [16].

TrustZone Trusted Language Runtime (TLR)[51] implements a .NET runtime that is isolated in TrustZone's secure world. Security critical parts of an application can be extracted into "trustlets" (similar to enclaves in SGX) which are executed within the TLR.

Apart from TrustZone as hardware the TCB includes the TLR implementation. As with all solutions with a software TCB at level *c* or broader, TLR is susceptible to privilege escalation if the TLR is compromised.

Though TrustZone could support trusted I/O, this feature is not available in the TLR, as it would require adding drivers to the TCB. TLR does not provide attestation. A remote party cannot verify the state of trustlets and the runtime it is interacting with. Thus, TLR does not strictly match the definition of trusted computing used in this thesis. It is listed as its implementation is interesting and comparable to other solutions.

Oasis[44] is comparable to SGX but avoids encrypting memory in DRAM. The concept hinges on using caches as RAM so that secrets never leave the CPU, e.g. are never stored on DRAM. To this end, Oasis adds a set of CPU instructions to "enable an isolated execution environment contained entirely on chip". As the authors themselves remark, Oasis is inferior to SGX in that it only supports applications of a very limited size.

Fides[57] uses a "small dynamic hypervisor to isolate [enclaves]" from the rest of the system. The hypervisor separates two virtual machines (VMs): the legacy and secure VM, similar to TrustZone. A minimal secure kernel isolates the different enclaves (called "self protecting module (SPM)") in the secure world.

The software TCB includes the hypervisor and secure kernel. A TPM is used to attest the hypervisor and security kernel state. The legacy kernel is excluded from the TCB. The "running legacy kernel is pulled in the legacy VM, and memory access control of both VMs is configured". This is possible using the dynamic TPM measurement features.

Attestation and data sealing are only available on the basic TPM level, which is bound to the overall system state. This means the hypervisor and secure kernel can be attested, and data can be sealed to this state. This cannot be done for individual modules.

TrustVisor[40] was developed by the authors of Flicker, with the goal of improving performance. It avoids slow TPM calls on the critical path by providing a virtual micro-TPM to each enclave (called "piece of application logic (PAL)"). With this micro-TPM, each enclave can be attested and perform data sealing.

The micro-TPMs are hosted by a trusted hypervisor, which is dynamically loaded and measured (as done by Fides). Thus the software TCB includes the hypervisor. The chain of trust when validating an enclave attestation is rooted in the TPM measurement. The chain thus includes the enclave, the micro-TPM and the hypervisor.

Unlike Fides, there is no secure kernel that isolates enclaves. This behaviour is emulated on a lower level by un-mapping enclave pages from the legacy operating system. Each enclave has its own virtual guest memory. Table 3.1 shows the software TCB as d, when really it only includes the hypervisor and enclave, but no operating system.

Flicker[41] enables fine-grained attestation and isolation of enclaves using only the dynamic attestation feature of a TPM and a supported CPU.

The intended use of the dynamic TPM measurement is to virtualise an untrusted operating system after booting it and lazily loading a privileged hypervisor (as described for Fides). To do so, the CPU enters a special execution mode to load the hypervisor with elevated privileges. During this time the legacy operating system is suspended. Its privileges are demoted to VM guest privileges. This way the untrusted operating system is effectively removed from the TCB.

Instead of loading a hypervisor Flicker executes the enclave during this special loading phase, which is called a "Flicker session". This session is also measured. After a cleanup phase (e.g. caches) regular execution is resumed and the result is returned.

This approach is nearly feature complete with regards to Table 3.1. It isolates enclaves on the same level as SGX. The hardware TCB only includes the TPM. The software TCB includes the enclave and only a small additional wrapper for handling parameter input/output and cleanup.

The main drawback of the approach is the performance. Slow TPM operations are on the critical execution path – they are executed every time the enclave is executed. Only one core is used and interrupts are disabled in the special CPU state. Thus the system is stalled for the duration of a Flicker session. For use in interactive systems, Flicker enclaves need to exhibit a very small runtime. This is diametrically opposed to TPM overhead incurred with each session. Only one Flicker session can be executed at any given point in time, as the special CPU mode is not intended for parallel use.

In summary, despite the apparent features and small TCB, Flicker is not well-suited for general-purpose applications due to its performance limitations.

3.3.2. Application Isolation

Microsoft Haven[8] uses SGX to isolate an entire legacy application within an enclave. Along with the application, a library operating system (Drawbridge LibOS[13]) is included in the enclave. "Drawbridge LibOS is a version of Windows 8 refactored to run as a set of libraries within the picoprocess."[14]

An additional shield module within the enclave mediates between the library operating system and the outside world (untrusted runtime). Any system call by the application is passed through the library operating system, secured by the shield module, and only then passed on through the untrusted runtime on to the untrusted operating system. The layers are depicted in Figure 3.6. This approach is re-visited in section 6.2 in the context of SCONE.[5]

Haven re-purposes SGX in a fashion. SGX was designed to isolate small security-critical parts of an application inside individual enclaves. This keeps the TCB small and can help when reasoning about security of the application. Haven tries to find a different solution to secure unmodified legacy applications.

This dramatically increases the size of the TCB but also provides additional benefits. The application must not be refactored or modified. In addition it protects against so-called Iago attacks by the operating system. A Iago attacks exploits the fact that an application may rely on a system call to be correctly executed instead of validating the results.

[13] `https://www.microsoft.com/en-us/research/project/drawbridge/`

[14] A picoprocess can interact with the operating system only through a very narrow system call interface. This is similar to the system call interface that hardware VMs use.

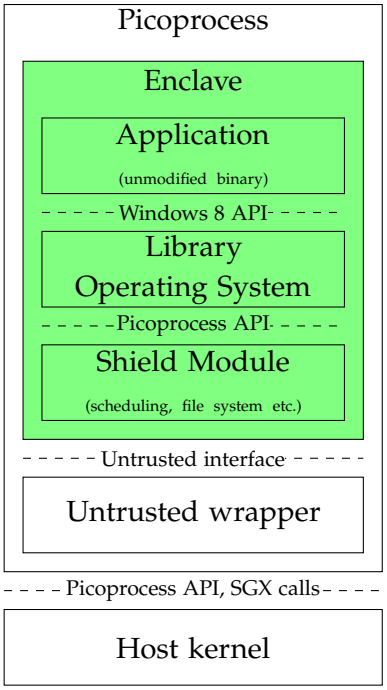


Figure 3.6.: Architecture of Microsoft Haven. The enclave (shaded green) isolates the entire unmodified application as well as a library operating system. Together with the shield module this protects the application from attacks by a malicious operating system. The enclave interacts with the host kernel through a the narrow picoprocess API, as the library operating system abstracts from higher-level system calls. The untrusted wrapper only passes on calls from and to the enclave. Reprinted as a simplified version from [8].

Haven is dated to 2014 and requires SGX, which is dated to 2015. Haven was implemented using SGX simulations and pre-release hardware before SGX-enabled CPUs became generally available.

Minibox[39] is comparable to TrustVisor, as it multiplexes the TPM into several virtual micro-TPMs. Minibox has a slightly different focus, as it aims to be a "two-way sandbox". Traditional sandboxing protects the execution environment, e.g. the operating system, from malicious applications. Minibox protects both the operating system and the application. Minibox executes applications in an isolated environment, called "Mutually Isolated Execution Environment (MIEE)".

A hypervisor provides isolation and the micro-TPMs. It is included in the TCB. The micro-TPMs enable data sealing and attestation on a per-application basis. A shield module in between checks and sanitises interaction in both directions.

InkTag[29] isolates applications in the same way as many other solutions in this section: through virtualisation. This is similar to Windows IUM. A hypervisor isolates the application, called "high-assurance process (HAP)", from the operating system. The hypervisor provides "secure files", which can be seen as a form of data sealing.

InkTag does not provide attestation, so a remote user cannot verify the state of the system. With regards to the definition used in this thesis, it is therefor not strictly a trusted computing solution. Technically, it is interesting and comparable to other solutions and thus listed.

A distinguishing feature of InkTag is its "para-verification". The InkTag hypervisor verifies the behaviour of the operating system. HAPs can check the verification status using hypercalls. To keep verification simple, InkTag requires the untrusted operating system to assist the hypervisor in its own verification.

Another interesting aspect is how isolation is technically achieved. InkTag does not rely on memory address translation as a hardware feature alone to isolate an HAPs memory. Instead, the hypervisor encrypts and hashes a HAPs memory pages on a context switch back to the operating system. This is somewhat comparable to SGX, where pages in DRAM are also encrypted. To describe it in the author's words: "InkTag uses hardware [memory management unit (MMU)] virtualisation for coarse-grained separation between secure and insecure data. Then it uses software only when needed, to manage the userspace portions of HAP page tables."

Overshadow[14] is comparable to InkTag. A hypervisor isolates applications. The implementation differs. Overshadow uses the terms "shadowing" and "cloaking". Memory is dynamically encrypted (cloaked) by the hypervisor depending on the "shadow context" accessing it. Only a cloaked application can read its own memory in decrypted form.

The hypervisor intercepts some system calls instead of passing them on to the untrusted operating system, such as file input and output. Files accessed by applications are memory-mapped. With the cloaking mechanism in place they are thus automatically encrypted when written to disk.

The idea of transparently encrypting file input/output is similar to Haven, where unmodified applications are protected. Overshadow also tries to set a low adoption barrier by minimising necessary changes to legacy applications.

The hypervisor has a single secret key which it uses for memory encryption. The key is also used to e.g. protect file meta data integrity when written to disk. This is somewhat similar to the memory integrity protection performed by SGX. However, SGX derives a unique key for every enclave (or enclave author). The file encryption in Overshadow cannot be counted as data sealing, as the data is not sealed to a specific application but encrypted with the "global" hypervisor key.

Like InkTag, Overshadow does not provide attestation and is, strictly speaking, not a trusted computing solution.

3.3.3. Operating System Isolation

CloudVisor[65] provides trusted VMs. VMs are an established deployment level in cloud environments. Users typically trust the cloud provider to execute a VM properly and properly isolate it from other VMs. This trust is not technologically grounded.

CloudVisor provides trusted VMs based on two factors. Firstly, VMs are protected from the hypervisor. This is implemented through nested virtualisation. A small security hypervisor in host mode controls the actual hypervisor. The security hypervisor is comparably small. It is dynamically loaded and attested through the TPM. It thus does not contain boot loader code. This reduces the size of the TCB. Secondly, the state of the security hypervisor can be remotely attested. A TPM is used for this. A user can then choose to release a VM image decryption key only to an attested hypervisor.

As for all trusted computing solutions that isolate on the virtualisation layer, the attack surface is large. If the guest operating system in a VM is compromised, all sensitive applications in that VM are compromised. Privilege escalation is also an issue. Also, CloudVisor does not protect against hardware attacks. Any party with hardware access can read the non-encrypted memory from DRAM (by tapping into the memory bus).

CloudVisor transparently inputs disk input/output. The data is not sealed to the VM state, but encrypted with a user-defined key. Only the hypervisor state can be attested, not the state of an individual VM.

Nova[56] is a micro-hypervisor, implemented from scratch. Using the same design principals as for micro-kernels, the Nova hypervisor is highly modularised. Its design follows the principle of least privilege. Only the bare minimum of Nova runs at the super-privileged VMM kernel level.

Nova is not a trusted computing solution. However, it showcases the principle of least privilege. This should be kept in mind when developing applications from Intel SGX.

NoHype[37] provides virtualisation without a hypervisor. Instead, resources are statically allocated: Each VM is allocated one CPU core and a slice of memory. However, VM

management must still take place. NoHype uses a management VM to load, start and stop other VMs. During execution, no interaction with a hypervisor is necessary.

NoHype does not run on standard hardware. It requires additional hardware virtualisation features that no CPU currently offers. NoHype limits a guest VM to a single core. It can parallelise across VMs, but not within VMs.

NoHype addresses VM isolation, but no further features such as data sealing or attestation. As attestation is not provided, NoHype is not a trusted computing solution as defined in this thesis. It is still listed due to its interesting approach.

vTPM[46] provides a virtual TPM to each VM. This TPM is designed to match the VM life cycle. It can be stored, loaded and migrated with its VM. The VM attestation provided by a virtual TPM is a compound attestation. The virtual TPM attests the VM. The hardware TPM attests the hypervisor and boot process.

TrustVisor and Minibox provide virtual TPMs at the module and application level. vTPM provides virtual TPMs at the VM level.

An interesting aspect is how migration is enabled. The virtual TPMs have to be linked to the hardware TPM so that the process is rooted in a hardware root of trust. If implemented naively, this would preempt the ability to later on migrate a virtual TPM to a different machine with a different hardware TPM. vTPM solves this using migrateable TPM storage keys, which the TPM standard defines.

Terra[22] is the first hypervisor-based solution for trusted computing. It introduced the idea of using a trusted hypervisor to isolate individual VMs. In its design, Terra – like vTPM – uses a hardware device for data sealing and attestation. It then exposes these features to every VM. The Terra prototype does not actually include such a hardware device. The authors identify a TPM as a good candidate.

3.4. Comparison

Table 3.1 shows a comparison of all trusted computing solutions presented so far. The table groups solutions by the TEE level they expose. Solutions with TEE level *a* allow the developer to isolate separate modules of his application. The narrower the TEE level of isolation is, the smaller the isolated parts can be. This makes them easier to verify and less likely to contain security bugs.

The TEE level controls the flexibility and ease of adoption. A broader TEE level may be more insecure, but can facilitate re-use of unmodified VMs or applications. Potentially, solutions higher up in the table can also emulate broader TEE levels. Haven shows how SGX, which isolates at module level, can be used to isolate an entire application including a library operating system.

Most solutions expose a narrow TEE level at the cost of a larger software TCB. TLR includes the secure kernel and .NET language runtime. Fides and TrustVisor include a hypervisor. Such a large software TCB is required when the underlying hardware does not support

General			Classification			Features		
Name	Reference	Year	TEE Level[a]	Software TCB[b]	Hardware TCB[c]	Attestation	Data Sealing	Parallelism
Sanctum	[16]	2015	a	a	custom CPU	✓		✓
Intel SGX	[42]	2015	a	a	SGX	✓	✓	✓
TLR	[51]	2014	a	c	TrustZone		✓	✓
Oasis	[44]	2013	a	a	custom CPU	✓	✓	✓
Fides	[57]	2012	a	d	TPM, Virt.	(✓)		✓
TrustVisor	[40]	2010	a	(d)	TPM, Virt.	✓	✓	✓
Flicker	[41]	2008	a	a	TPM	✓	✓	
Windows IUM		2016	b	d	Virt.			✓
Haven	[8]	2014	b	c	SGX	✓	✓	✓
MiniBox	[39]	2014	b	d	TPM, Virt.	✓	✓	✓
InkTag	[29]	2013	b	d	Virt.	(✓)	✓	✓
Overshadow	[14]	2008	b	d				✓
CloudVisor	[65]	2011	c	d	TPM, Virt.	(✓)		✓
Nova	[56]	2010	c	d	Virt.			✓
NoHype	[37]	2010	c	c	custom CPU			(✓)
vTPM	[46]	2006	c	d	TPM, Virt.	✓	✓	✓
Terra	[22]	2003	c	d	TPM, Virt.	✓	✓	✓
ARM TrustZone	[4]	2003	c	c	TrustZone			✓
TPM	[59]	2002	d, e	d, e	TPM	✓	✓	✓

Table 3.1.: Comparison of trusted computing solutions. Rows are ordered first by *Trusted Execution (TEE) Level*, then by *Year*. The columns *TEE Level* and *Software Trusted Computing Base (TCB)* refer to Figure 3.2. All commercial solutions can be used stand-alone (shown in their own row). Most commercial solutions are also used as hardware foundation by solutions from research (shown in the *Hardware TCB* column). Solutions that do not support attestation cannot strictly be considered implementations of trusted computing. They do not support the verification step in Figure 2.6.

[a] The software levels a developer must provide to use the solution. E.g. *c* means that an operating system and the application must be provided. This value of this column is automatically the lower bound for the value of *Software TCB*.

[b] The software levels that are included in the solution's TCB. E.g. *d* means the entire virtualisation stack is included in the TCB. The software TCB is the union of software levels that the solution internally adds and the software levels the developer must add (*TEE Level*).

[c] *Virt.* stands for hardware virtualisation support. *SGX, TrustZone, TPM* refer to the respective commercial solutions.

isolation at the desired level. Solutions with a smaller software TCB require specialised hardware. This usually means a larger hardware TCB. Shifting the TCB from software into hardware is not necessarily an improvement. Firstly, it is hard to draw a clear line between the two. SGX is considered a hardware feature, but is implemented mostly in micro-code, the firmware of the CPU.[15] Secondly, a hardware implementation must not automatically be more secure than the alternative in software.

All presented solutions utilise the CPUs processing power. A TPM is used as an external secure element by some. This is only responsible for attestation and handling of cryptographic keys. Some solutions such as Flicker and NoHype do not make full use of the CPUs processing power.

This is to the author's knowledge the first comparison of its kind. A comparison of security features of some solutions is presented in [15].

4. Intel SGX

This chapter describes Intel Software Guard Extensions (SGX) in more detail. Costan et al. provide an exhaustive, in-depth description and analysis of SGX which is referred to as additional reading material.[15] This chapter briefly describes the basic concepts of SGX and then summarises further findings from research. This includes performance studies, known criticism and security issues, as well as noteworthy applications built on top of SGX.

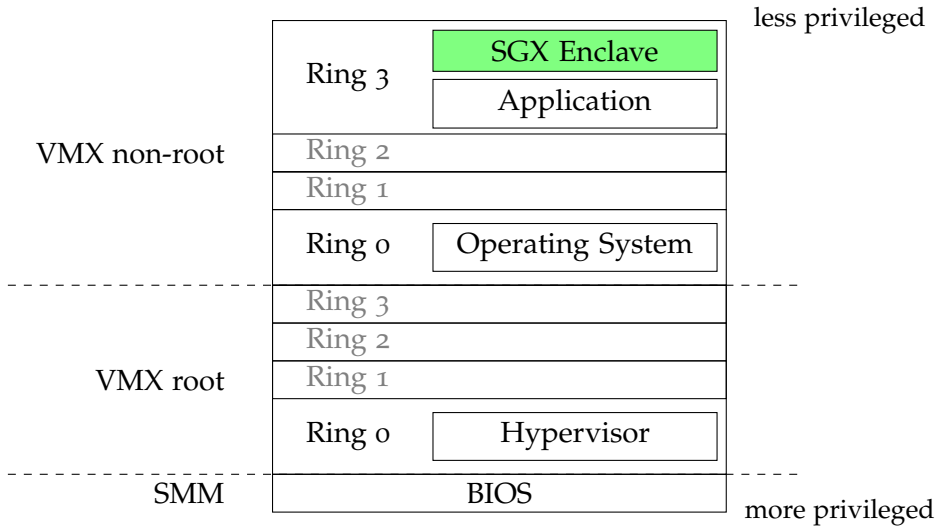


Figure 4.1.: Intel SGX enclave within the privilege level hierarchy. An Intel CPU typically has two privilege schemes. Privilege rings are the oldest concept, of which nowadays only ring zero and three are used to separate the operating system and applications. These are often called kernel and user mode. Virtualisation support adds another privilege scheme. The hypervisor runs in VMX root mode. It is protected from the guest VMs running in VMX non-root mode. The BIOS runs at the highest privilege level in system management mode (SMM). SGX enclave mode adds an inverse isolation layer. The two existing privilege schemes protect more privileged components (bottom) from less privileged ones (top). SGX enclaves are in the least privileged layer, but are protected from all more privileged components. Reprinted from [15].

4.1. Overview

Intel SGX is a trusted computing solution. It is fully contained within the CPU and is exposed as an instruction set. As described in chapter 3, SGX protects individual software modules in so-called "enclaves". Compared to other solutions, the TCB is small. It includes only the

protected module and the CPU.[15] SGX allows remote parties to verify the state of an enclave (attestation). It provides additional features, such as data sealing, on top.

The operating system, system management code and other parts of the application do not have to be trusted. The enclave is also protected from code running in system management mode (SMM), as well as from direct memory access (DMA).[42] SGX changes the memory access semantics by introducing a protection scheme inverse to the existing privilege levels.[15, ch. 6.2]. Figure 4.1 shows how enclaves relate to existing privilege levels.

Figure 4.2 shows an abstract view of an application's address space layout. The enclave's memory is protected by the CPU from direct access by any component but the enclave. When enclave memory is loaded into the CPU (caches), the CPU can enforce isolation by checking whether it is currently executing code of the correct enclave. If a memory page leaves the control of the CPU (when writing it to DRAM) it is encrypted and integrity-protected.[26] More details on SGX's memory management are given in section 4.3.

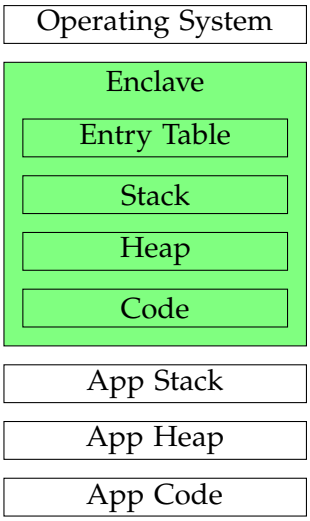


Figure 4.2.: Application address space with an Intel SGX enclave. The CPU only allows access to the enclave's memory if it is currently executing code belonging to that very same enclave. An enclave can be entered only at specific points in the code, defined in the entry table. The entire enclave memory (including code and entry table) is measured when the enclave is initialised. The CPU can attest to a remote party that it loaded the enclave correctly. Reprinted from [42].

An interesting aspect of SGX is that it relies on the untrusted operating system to perform its regular management tasks such as scheduling and memory allocation. This includes the steps for setting up an enclave. Enclave attestation would expose any attempts by a malicious operating system to load a tainted enclave. Costan et al. put it like this: "SGX design expects the system software to allocate the EPC pages to enclaves. However, as the system software is not trusted, SGX processors check the correctness of the system software's allocation decisions." [15] Figure 4.3 shows the enclave life cycle.

This reliance on the untrusted operating system keeps the SGX implementation small. It does however open up certain attack avenues. A denial of service (DoS) attack is straight-

[15]The TCB also includes Intel's architectural enclaves.

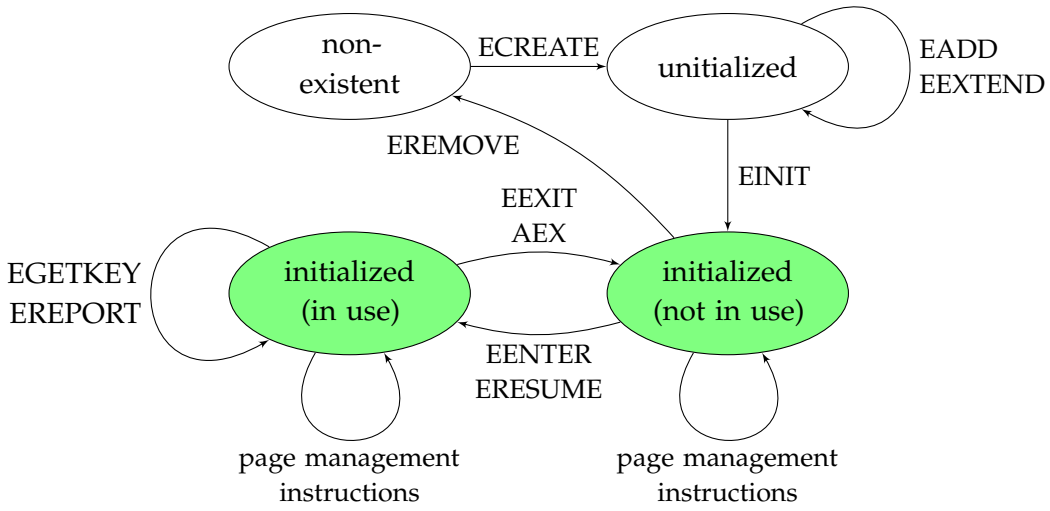


Figure 4.3.: Intel SGX enclave life cycle. The enclave's memory is protected in states shaded green. State transitions occur when CPU instructions are executed. E.g. *ECREATE* creates a new, uninitialised enclave. The operating system is expected to load the enclave by adding pages and extending the measurement of the enclave (similar to TPM measurement in Figure 3.3). Once *EINIT* is called, the enclave is locked down. Its measurement is final and does not change when the enclave executes and changes internal data. The OS can no longer access the enclave's memory pages. The enclave can now only be entered via *EENTER* (and after interrupts through *ERESUME*) at locations defined in the entry table. *EGETKEY* and *EGETREPORT* use the initial measurement (the enclave's *identity*) for attestation and to derive cryptographic keys data sealing. The *page management instructions* refer to paging into and out of Enclave Page Cache (EPC), a special memory area. Reprinted as a simplified version from [15].

forward, as the operating system can refuse to schedule any enclave threads. In the context of remote computation, the infrastructure owner could choose to cut the power at any time, so this is not really a disadvantage. More serious security issues are described in section 4.4.

Each SGX-capable CPU has an embedded cryptographic private key. Using a special group signature scheme, the CPU uses this key to attest the state of an enclave.[34] Attestation can occur locally to setup secure communication channels between different enclaves on the same CPU.[1] It can also occur remotely. In this case, the attestation is not performed purely in hardware, but relies on additional so-called "architectural enclaves".[15] These enclaves increase the size of the software TCB. They are also the main source for criticism of SGX as explained in section 4.4.

Code running in an enclave may not execute certain calls. These can only be handled by the untrusted wrapper.[33] Among them are instructions which may cause a *VMEXIT*[16], input/output instructions[17] and instructions which require a change in privilege levels (e.g. system calls).[18][31] It is still possible to securely communicate with enclaves using a key exchanged during the attestation process. An enclave can use a key derived from its identity (initial measurement) to encrypt any data it wishes to expose to the untrusted world.

Multiple threads can be active at the same time in an enclave.[42] The number of threads must be statically defined before the enclave is initialised. Also, the maximum enclave size must be fixed before initialising the enclave.[19] SGX capable CPUs are available since the end of 2015.[20].

4.2. Enclave Development

Intel offers a Software Development Kit (SDK) for authoring enclaves and integrating them into an application. The SDK is available for both Windows[21] and Linux[22] The SDK provides the following features:[32, 31]

C and C++: These are the only programming languages supported by the SDK.

Interface definition: An enclave's interface is defined in the Enclave Definition Language (EDL). This is described in more detail later on.

Debugging: This is actually a SGX hardware feature. An enclave in debug mode is not protected by the CPU.

Simulation mode: In the absence of SGX hardware, the hardware is simulated for development purposes.

[16] CPUID, GETSEC, RDPMC, RDTSC, RDTSCP, SGDT, SIDT, SLDT, STR, VMCALL, VMFUNC

[17] IN, INS/INSB/INSW/INSD, OUT, OUTS/OUTSB/OUTSW/OUTSD

[18] Far call, Far jump, Far ret, INT n/INTO, IRET, LDS/LES/LFS/LGS/LSS, MOV to DS/ES/SS/FS/GS, POP DS/ES/SS/FS/GS, SYSCALL, SYSENTER

[19] SGX version 2 allows a dynamic number of threads and dynamic memory size. No hardware is available at the time of writing.

[20] `https://github.com/ayeks/SGX-hardware`

[21] `https://software.intel.com/en-us/sgx-sdk`

[22] `https://github.com/01org/linux-sgx`, open source under the very liberal BSD license.

Trusted library: Helper functions for enclave development. This includes a subset of the standard C library (e.g. without file input/output), random number generation, cryptographic primitives, key exchange and data sealing.

Complete authoring chain: Enclaves can be compiled and signed so that they could be loaded in production use. See section 4.4 for the restrictions that apply.

An example EDL interface definition is shown in Listing 4.1. It is divided into a trusted (E-call) and untrusted (O-call) section. Based on this interface, the SDK generates proxy functions. For all *trusted* functions (E-calls) proxies are generated for the untrusted wrapper. For all *untrusted* functions (O-calls), proxies are generated for the enclave.

The proxy code is necessary for parameter marshalling. The function signature includes additional annotations for the parameters. The annotations show the direction of data flow (*in*, *out*, *user_check*). If *in* (and/or *out*) are specified, the proxy code will copy the parameter by value before calling the function (and/or afterwards). A pass-by-reference can be achieved with *user_check*. Pass-by-value is recommended for security reasons. The enclave cannot rely on untrusted memory to be stable. However, copying and checking parameters adds overhead. This is discussed in section 4.3.

Listing 4.1: enclave.edl[23] – Enclave Definition Language (EDL) example file. EDL is used by the Intel SGX SDK to specify an enclaves interface on the function level. The enclave's entry table is generated based on the trusted section of the EDL file. This EDL file defines only E-calls, but no outgoing (untrusted) O-calls. The example is taken from the demo consumer of the author's SGX helper library.

```
1  enclave {
2      trusted {
3          /* add secret to sealed file */
4          public void add_secret(int secret);
5          public void print_secrets();
6          public void test_encryption();
7          public void set_key([in, size=128] uint8_t *key);
8      };
9
10     from "../sgx-lib/sgx_lib_t/sgx_lib.edl" import *;
11     untrusted {
12
13     };
14 };
```

The proxy needs to know how much data to copy for pointer arguments. This is handled by the annotations *size*, *sizefunc* and *count*. The first two define the size of an individual element statically or dynamically. The number of elements can be defined with *count* either statically as a number or dynamically by referencing a different scalar parameter. For a full reference of EDL, see [32].

The **from ... import** in Listing 4.1 also shows how EDL files can be composed. In this case, library helper functions are included. This is also the method of choice for adding

[23] https://github.com/ftes/sgx-lib-consumer/blob/thesis/enclave/enclave.edl

remote attestation and key exchange to an enclave.[24] The helper library in question assists in prototyping SGX enclaves and is described in chapter 5.

In addition to the architectural enclaves (attestation etc.), Intel also provides some helper enclaves as part of the Platform Software (PSW). These enclaves expose functionality such as monotonic counters and trusted time.[32] They can be accessed via trusted library functions included in the SDK. These enclaves rely on the Manageability Engine (ME), which is a part of Intel CPUs, to provide these features.[25]

4.3. Performance

In principle, the CPU's full processing speed is available in SGX enclaves. This is an advantage over solutions with external secure elements. However, several factors have a observable performance impact on enclave performance. Isolation is achieved by protecting an enclave's memory. The additional memory layers introduced to enforce this isolation have an impact on access speed. Using Intel's SDK on the other hand apparently results in a larger performance impact. Existing findings from research are now presented.

Figure 4.4 shows what performance overhead an enclave has on memory access. Prefetching hides most of the performance impact for sequential reads and writes.[5] Random reads and writes highlight the actual performance impact. Two major factors impacting access times can be identified in the diagram.[26][5]

L3 cache size: Enclave memory remains decrypted within the CPU's caches. As long as all enclave memory fits in the L3 cache, memory access times are roughly equal. If the cache is exceed, pages must be fetched from DRAM, decrypted and integrity-checked.

Enclave Page Cache (EPC) size: The EPC is a special section of DRAM. Pages that do not fit into EPC must be paged out to regular sections of DRAM. The EPC size is limited to 128MB on current SGX CPUs, of which 92MB can be used by user's enclaves. The rest is needed for meta data and Intel's architectural enclaves.

This performance overhead means that enclave memory is a valuable resource and must be managed accordingly. If possible, the combined size of all enclaves on a system should remain beneath the magical 92MB limit to avoid the 1000x performance penalty. Even better, the size of the L3 cache should not be exceeded.

The SDK provided by Intel should also be used with caution regarding performance. The SDK introduces the concept of E-calls and O-calls, which are synchronous transitions into and out of the enclave.[33] The SDK's performance is evaluated in [5]. They compare different solutions for executing system calls from within an enclave. The first option is to use the untrusted wrapper as a synchronous proxy (E-call for every system call). The

[24] https://github.com/01org/linux-sgx/blob/1115c195cd60d5ab2b80c12d07e21663e5aa8030/SampleCode/RemoteAttestation/isv_enclave/isv_enclave.edl

[25] https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/607330

[26] [31] also lists these as performance bottlenecks.

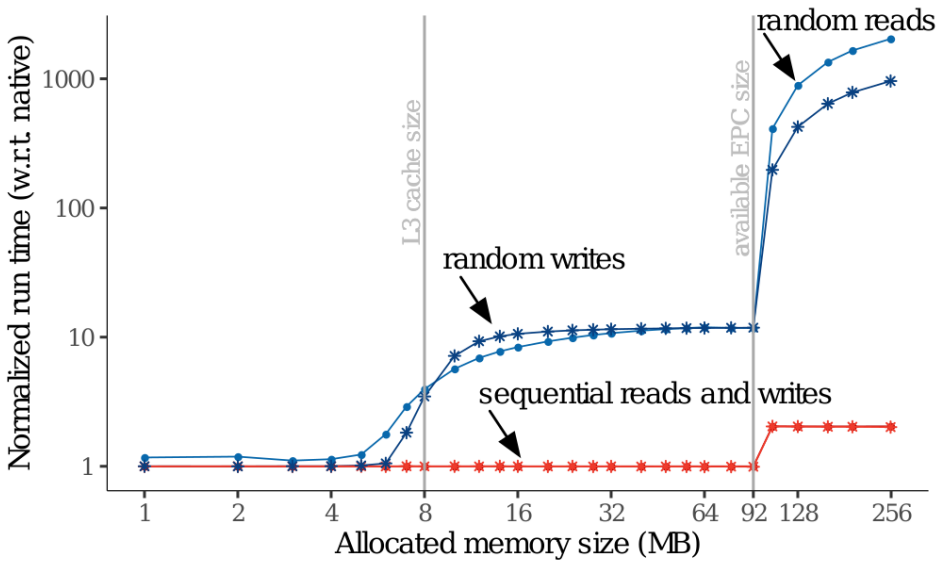


Figure 4.4.: Memory access speed in Intel SGX enclaves. Access times are normalised w.r.t native (non-enclave) access. The two limiting factors, L3 cache size and Enclave Page Cache (EPC) size are shown as grey lines. Sequential access hides some of the overhead due to pre-fetching. Reprinted from [5].

CPU must switch execution context, execute the system call, and pass the result back to the enclave. This adds a 10x overhead for file input/output. Better performance is achieved with an asynchronous executor thread pool outside of the enclave. This solution results in performance comparable to native execution.[5] However, the Intel SDK can apparently not be used to build this complex kind of interaction.

4.4. Known Criticism

Intel SGX is a technically exciting solution for trusted computing. Related to SGX, criticism has been voiced on multiple levels. It ranges from critique of trusted computing in general, debatable SGX design decisions up to security bugs.

Trusted computing in general: The release of a commercially available solution for trusted computing has re-triggered an existing debate. Intel SGX protects enclaves from any access by the operating system and hardware owner.

Depending on the standpoint this may be a desirable feature or an intrusion into personal rights. When deploying an application to the cloud, it may be desirable for the software vendor to keep certain data secret from the infrastructure provider and other tenants. When developing a blu-ray player (see section 4.5), it may be desirable for the blu-ray industry to keep decryption keys secret and guarantee that digital rights are not infringed. When executing an application as an end user or infrastructure provider, it may be desirable to be in full control of the application.

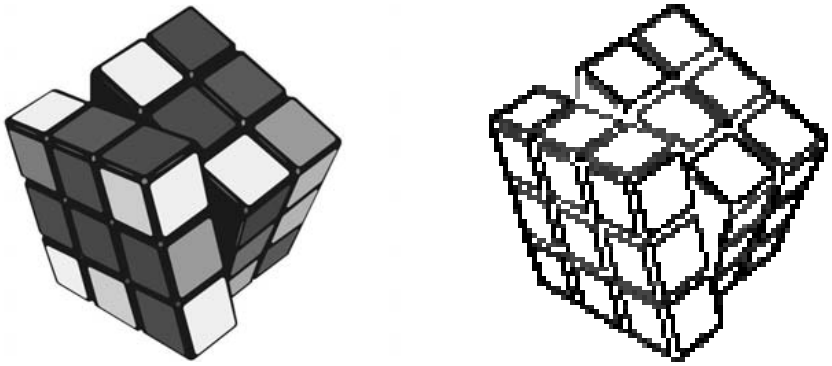
SGX turns the tables: the software vendor can – to a certain degree – take control of the hardware without interference of the hardware owner. This gives cause to debates of ownership.

Malware in enclaves: Enclaves are protected from the operating system and hardware owner. This can also be a disadvantage from a security standpoint. Malware protected in an enclave is an often stated example. Two factors are in place that should prevent this. Firstly, enclaves cannot perform any input/output, so part of the malware would have to live in an (observable) unprotected wrapper.[15, ch. 6.8] Secondly, Intel can decide which enclave software will be loaded by an Intel CPU (white labelling). This is again a cause of criticism.

Intel only white labels the identity of an enclave – its initial measurement. Because enclave code may be self-modifying it is possible to dynamically load encrypted malicious code into an enclave. This is a viable attack vector if an exploit is found for a white labelled enclave, into which malware could then be loaded.[54] AsyncShock is a tool that can help exploit enclave bugs. It targets synchronisation bugs in enclaves. Using such a bug AsyncShock helps to extract secrets or modify the control flow in that enclave. Enclaves approved by Intel could then be misused.[62]

Side-channel attacks: The following attacks have been successfully identified. The first two have been successfully carried out.

- Memory access pattern analysis of well-known libraries in an enclave. This is done with the help of the operating system, which simulates page faults to detect memory access. As a result, images processed by a library within an enclave were re-constructed. Oblivious RAM techniques and address space layout randomisation are proposed as counter-measures.[63]
- Cache Prime+Probe attack on co-located enclaves. Based on knowledge of instruction execution times, the authors could measure memory access times from within an enclave and deduce cached values. With this technique an RSA key was extracted from another enclave running a standard RSA implementation. The authors propose several countermeasures. This attack highlights the problem of having protected malware inside an SGX enclave.[54]
- Hyper-threading execution timing. SGX does not prevent the use of hyper-threading. If an enclave shares a logical processor with a snooping thread that thread could find out what instructions the enclave is executing as well as its memory access patterns. The authors propose to disable hyper-threading. Also the hyper-threading status should be included in the enclave measurement so that it can be attested.[15]



(a) original **(b)** recovered

Figure 4.5.: Result of a side-channel attack on SGX. By analysing the memory access pattern of an enclave running the image processing library *libjpeg*, certain features of the input image could be reconstructed. Reprinted from [63].

Vulnerability of architectural enclaves: SGX mainly consists of hardware: CPU microcode and memory encryption engine. To make it feature-complete and usable, Intel adds some architectural enclaves (launch enclave, quoting enclave) and platform software (PSW). This keeps the hardware TCB small at the cost of a larger software TCB. The quoting enclave access the CPU's attestation key. A bug in the quoting enclave could expose that attestation key. Intel's EPID group signature scheme allows revocation of individual keys. But a quoting enclave bug would potentially expose the attestation keys of all SGX CPUs until a patched quoting enclave is deployed.[34] Intel is thus under constant pressure to keep it's architectural enclaves secured against any new attacks. An exploit for an architectural enclave would also allow malware to be loaded into a protected enclave as described earlier.

Intel controls enclave launch: Enclave launching, like attestation, is also implemented in software in an architectural enclave. This lets Intel control which enclaves may be launched. An enclave can be loaded either if it is authored by Intel (e.g. the launch enclave) or if the launch enclave grants the launch. Based on Intel's patents for SGX, Costan et al. surmise that "the Launch Enclave is intended to be an enclave licensing mechanism that allows Intel to force itself as an intermediary in the distribution of all enclave software".[15]

This allows Intel to prevent malicious software from being loaded. This also allows Intel to control which software is loaded in general. The benefits for Intel from a business standpoint are obvious. This level of control over an end-users hardware can be seen as "software security equivalent to the Net Neutrality debates".[15]

4.5. Applications

Leaving the (well-founded) criticism behind, SGX has the potential to be the foundation for innovative applications. This section highlights some interesting SGX-based applications from research. It does not describe such applications that themselves can be considered frameworks or infrastructure layers for securing other applications such as SCONE or Haven. These are discussed in section 6.2.

Proof of elapsed time: Bitcoin is the prototypical ledger-based crypto-currency. Its security revolves around the notion of proof-of-work. As long as the assumption holds that the majority of processing power in the Bitcoin network belongs to honest users, the majority of the network eventually behaves as expected.[43]

The proof-of-work to be brought forward in Bitcoin is the solution of a hash puzzle. Participants (miners) must hash a fixed input combined with an input of their choice so that the resulting hash satisfies a certain criteria. The first miner to find a solution to the puzzle wins. Competing in the network requires investment of processing power and thus money.

The downside is that this processing power is invested in finding a solution to a random puzzle. This solution has no inherent value outside of Bitcoin. Alternative crypto-currencies such as Primecoin use puzzles with solutions that have an inherent value, such as finding new prime numbers.[38]

A different approach based on SGX enclaves is proposed in Intel's Sawtooth[27] project. Sawtooth introduces proof-of-elapsed-time (PoET) as an alternative to proof-of-work. Essentially, Intel CPUs are used as an attestable source of true random numbers. If this number is viewed as a wait time, the participant to generate the lowest wait time wins.[30]

Digital rights management (DRM): The current version of Cyberlink's PowerDVD requires SGX hardware for playback of ultra high definition (UHD) blu-rays.[17] This is an example of how SGX can be used on consumer devices rather than cloud infrastructure. Cyberlink does not explain what SGX is used for. It is likely that an enclave handles the decryption of the blu-rays content. The decryption key would then only provided to attested enclaves by the Cyberlink server.

Secure ZooKeeper: ZooKeeper[28] is a key-value store used to provide configuration, naming, synchronisation etc. in distributed applications (e.g. micro-service architectures). SGX can be used to harden existing applications, which is the topic of this thesis. SecureKeeper uses enclaves to protect the data managed by ZooKeeper. When the data is stored outside of the enclave, e.g. on disk, it is encrypted. The Java native interface (JNI) is used to bind the Java implementation of ZooKeeper to the enclaves.[12]

Secure Hadoop map-reduce: VC3 is a prototype of Microsoft Research that "runs distributed MapReduce computations in the cloud while keeping their code and data secret."

[27] https://intelledger.github.io
[28] https://zookeeper.apache.org/

Hadoop is used as the underlying map-reduce engine. The map and reduce jobs run within enclaves. All other software components such as Hadoop or the operating system are kept outside of the TCB. Enclave code, input data and the results remain encrypted when outside of enclaves. VC3 achieves full Hadoop compatibility by performing all setup steps "in-band" as map-reduce jobs: distributing enclave code, performing attestation, and distributing data decryption keys.[53]

4.6. Conclusion

SGX is the first trusted computing solution that is likely to see widespread adoption. It is shipped with many current Intel desktop processors[29]. SGX has potential use-cases for both end-consumer devices and cloud infrastructure. Its main advantage when compared to other solutions is the achievable processing speed. SGX provides module-level TEEs and has an small overall TCB.

There are many potential security issues and weighty criticism regarding Intel's design decisions and the influence Intel thus has over end-user hardware. Only time can tell how well SGX as a technology will be received and whether it can live up to its expectations. The continued security of the architectural enclaves seems to be a crucial factor.

[29] https://github.com/ayeks/SGX-hardware

5. Intel SGX Helper Library

The case studies conducted for this thesis use the Intel SGX SDK for Windows.[30] To make prototyping faster and easier, a helper library wrapping the SDK was developed alongside the case studies. The library contains scripts and wrapper functions that make working with the SDK easier. The library also assists in constructing a shim C library (Figure 6.1). The concepts of this library and some usage guidelines are introduced in this chapter.

The full code is not printed in this chapter or the appendix. Please refer to the Git repository.[31] The repository also contains more in-depth details on configuration and usage. A demo consumer project showcases usage of the library.[32] The Git tag `thesis` in these repositories marks the commit from which the code listings in this thesis are taken.

No SGX hardware was available at the time of implementation. The library is only usable for simulation mode. It can not be used in production-ready enclaves.

The library is split into a trusted and untrusted module. The consumer can include the header files and link against these library modules. Also, the consumer must include the library's EDL file in his.

The library tries to help with four aspects of enclave development:

Generate O-call proxies: This is necessary if the C library lives outside of the enclave. Then a shim is needed inside the enclave to proxy calls to the outside (see Figure 6.1). Defining these proxies involves touching several files and repeatedly inserting a similar method signature. The `add_ocall.sh` script speeds up this process. See Table 5.1 for details on how to use this script.

As an example, consider adding a proxy for the `_ftelli64`[33] Windows C library function. The helper script has to be called as shown in Listing 5.1.

Listing 5.1: Example invocation of O-call generation script. The script generates EDL code, trusted header code and trusted and untrusted proxy implementations. The environment variables do not have to be set. Their default values correspond to the directory and file layout of the library.

```
1  sgx-lib/add_ocall.sh "int64_t _ftelli64([user_check] FILE* file);"
```

The following listings show the code generated by the script. Figure 5.1 shows how the generated code interacts with the SDK and C library.

[30] At implementation time (first half of 2016), the Linux SDK was not yet available.
[31] `https://github.com/ftes/sgx-lib/tree/thesis`
[32] `https://github.com/ftes/sgx-lib-consumer/tree/thesis`
[33] `https://msdn.microsoft.com/de-de/library/0ys3hc0b.aspx`

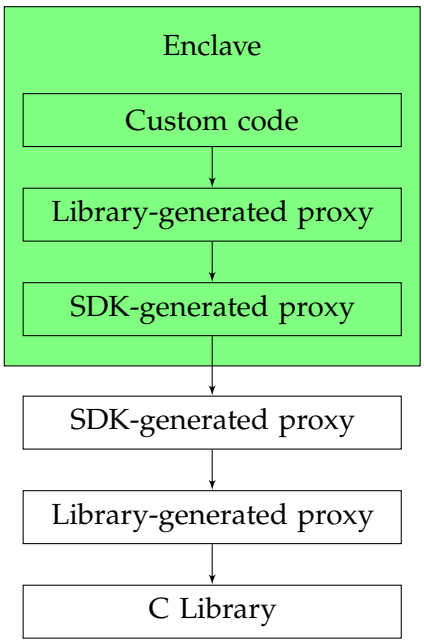


Figure 5.1.: Interaction of proxies generated by Intel's SDK and the helper library. The SDK proxies deal with parameter handling. Depending on the EDL definition, parameters are checked and copied back and forth. The enclave library proxy checks the return value and prints human readable error messages. The untrusted library proxy delegates to the C library.

Listing 5.2: sgx_lib.edl (extract).[34] The EDL interface definition is extended with the O-call. This does not affect the enclave entry table, as this only controls the allowed E-calls.

```
46  /* GENERATE OCALL CODE AFTER THIS LINE */
47  int64_t _ftelli64_ocall([user_check] FILE* file);
```

Listing 5.3: sgx_lib_t_stdio.h (extract).[35] The trusted header file is modified to include the proxy function's signature.

```
41  /* GENERATE OCALL CODE AFTER THIS LINE */
42  int64_t _ftelli64(FILE* file);
```

Listing 5.4: sgx_lib_t_stdio.c (extract).[36] The trusted proxy implementation. The proxy also acts as an adapter. It converts the O-calls signature (with the return value passed as a pointer) to the original signature. The proxy checks for errors. If an error is encountered, a meaningful error description is printed using an O-call.

```
190  /* GENERATE OCALL CODE AFTER THIS LINE */
191  int64_t _ftelli64(FILE* file) {
192    int64_t ret;
193    check(_ftelli64_ocall(&ret, file));
```

[34] https://github.com/ftes/sgx-lib/blob/thesis/sgx_lib_t/sgx_lib.edl
[35] https://github.com/ftes/sgx-lib/blob/thesis/sgx_lib_t/include/sgx_lib_t_stdio.h
[36] https://github.com/ftes/sgx-lib/blob/thesis/sgx_lib_t/sgx_lib_t_stdio.c

```
194   return ret;
195 }
```

Listing 5.5: sgx_lib_u_stdio.c (extract).[37] The untrusted proxy implementation. This simply delegates to the C library implementation.

```
35 /* GENERATE OCALL CODE AFTER THIS LINE */
36 int64_t _ftelli64_ocall(FILE* file) {
37   return _ftelli64(file);
38 }
```

The untrusted wrapper seems superfluous. Rather, the SDK could directly be linked to the C library implementation. The SDK supports this feature by adding `[cdecl, dllimport]` to a function signature in the EDL file.[32] However, the generated stub in the enclave has a different signature in case the function has a return value. The generated signature of the trusted `fopen` O-call is shown in Listing 5.6.

Listing 5.6: Generated O-call signature for standard C library function. The SDK passes the return value via a pointer parameter.

```
void fopen(FILE* retVal, const char* filename, const char* mode);
```

To use unmodified legacy code in an enclave, the library functions must have the exact same signature. To provide trusted functions with the original signature, one has to overload the functions in the enclave. This is not possible in C. The library is written in C to facilitate usage in both C and C++ projects. As a workaround the O-calls are appended with a `_ocall` suffix. No overloading is thus necessary. Instead, an untrusted proxy implementation is generated which delegates to the C library implementation.

Translate error codes to messages: A variety of error codes is defined for SGX[38]. Many SDK functions and the generated proxies can return these error codes. Manually looking up their meaning is time-consuming.

The library contains a trusted[39] and untrusted[40] utility function to check the return value. The descriptions are scraped from the Intel SDK's `sgx_error.h`[41] file. The scraping script is included as part of the library. It has to be re-executed in case the error codes or messages change. For usage details, see Table 5.1.

Developer-friendly encryption: The SDK includes a cryptography library. It can also seal data to an enclave's identity (see section 2.2). However, some of the SDK's cryptography functions are cumbersome to use. Due to the use of block ciphers and nonces the encrypted/sealed data size is not trivial to determine. The library provides a thin

[37] https://github.com/ftes/sgx-lib/blob/thesis/sgx_lib_u/sgx_lib_u_ocalls_stdio.c
[38] https://software.intel.com/en-us/node/709252
[39] https://github.com/ftes/sgx-lib/blob/thesis/sgx_lib_t/sgx_lib_t_util.c#L10
[40] https://github.com/ftes/sgx-lib/blob/thesis/sgx_lib_u/sgx_lib_u_util.c#L8
[41] The version included in the Windows SDK is probably identical to the Linux version: https://github.com/01org/linux-sgx/blob/sgx_1.9/common/inc/sgx_error.h

wrapper for data sealing (the SDK's interface is simple enough). The library adds a more extensive wrapper for encryption. Regular encryption must be used instead of data sealing if the developer needs to be in control of the encryption key. This can be the case if encrypted data is provided as an input, and not encrypted by the enclave itself. Also, encryption adds far less overhead than the data sealing performed by the SDK as shown in chapter 7.

Listing 5.7 shows the corresponding functions exposed by the library.

Listing 5.7: sgx_lib_t_crypto.h (extract).

```
18 uint32_t get_sealed_data_size(uint32_t plaintext_data_size);
19 int seal(const void* plaintext_buffer, uint32_t plaintext_data_size,
       ↪ sgx_sealed_data_t* sealed_buffer, size_t sealed_data_size);
20 int unseal(void* plaintext_buffer, uint32_t plaintext_data_size, sgx_sealed_data_t
       ↪ * sealed_buffer);

27 uint32_t get_encrypted_data_size(uint32_t plaintext_data_size);
28 int encrypt(const void* plaintext_buffer, uint32_t plaintext_data_size,
       ↪ sgx_lib_encrypted_data_t* encrypted_buffer, sgx_aes_ctr_128bit_key_t* key);
29 int decrypt(void* plaintext_buffer, uint32_t plaintext_data_size,
       ↪ sgx_lib_encrypted_data_t* encrypted_buffer, sgx_aes_ctr_128bit_key_t* key);
```

Encryption/decryption is done using AES block cipher in counter mode (`sgx_aes_ctr_encrypt` library function). According to NIST, counter mode encryption is efficient because output blocks can be derived in parallel, even before the complete payload is available.[42][13] NIST also mandates that the counter must be unique over all messages encrypted under the same key. If the counter space is large enough compared to the payload sizes, the encryption key can be re-used if the initial counter – also known as initialisation vector (IV) or nonce – is chosen at random.

The library `encrypt` function chooses a random IV using SGX's trusted source of randomness by calling `sgx_read_rand`.[43] The IV is added to the encrypted output. The `decrypt` function does the opposite: It reads the nonce from the beginning of the input data and uses it to decrypt the data.

Transparently encrypt input/output: The concept of transparent de- and encryption of input/output data is used in related work. This protects data operated on by legacy code without any code modifications. The library supports this concept by intercepting calls to the C library for file input/output. Replay protection is *not* added. The enclave will not notice whether the most recent or an older value is provided.

The developer can choose the desired security level at compile time using macros:

- No security. Useful during development, file input/output happens in plain text.

[42] The library does not make full use of this fact for decryption, because the data is first copied into the enclave in full by the SDK proxy.

[43] `https://github.com/ftes/sgx-lib/blob/thesis/sgx_lib_t/sgx_lib_t_crypto.c#L115`

- Encryption with custom key. Useful for debugging. A symmetric encryption key is required, which can be set using `set_secure_io_key()`[44].
- Data sealing. This is the default option and seals all input/output to the enclaves identity.

Listing 5.8 shows the preprocessor macros that switch the behaviour.

Listing 5.8: sgx_lib_t_stdio.h (extract).[45] The macros `SGX_INSECURE_IO_OPERATIONS` and `SGX_SECURE_IO_OPERATIONS_KEY` control how input/output is protected. By default, it is sealed.

```
27 #ifdef SGX_INSECURE_IO_OPERATIONS
28 #define fwrite fwrite_insecure
29 #define fread fread_insecure
30 #else
31   #ifdef SGX_SECURE_IO_OPERATIONS_KEY
32   void set_secure_io_key(sgx_aes_ctr_128bit_key_t key);
33   #define fwrite fwrite_encrypted
34   #define fread fread_encrypted
35   #else
36   #define fwrite fwrite_sealed
37   #define fread fread_sealed
38   #endif
39 #endif
```

[44] https://github.com/ftes/sgx-lib/blob/thesis/sgx_lib_t/sgx_lib_t_stdio.c#L37
[45] https://github.com/ftes/sgx-lib/blob/thesis/sgx_lib_t/include/sgx_lib_t_stdio.h

Name	Trusted	Untrusted	Details
`add_ocall.sh`[a]	✓	✓	Script that generate an O-call. Use e.g. for generating C library proxies. Writes EDL definition, trusted and untrusted wrapper code. A *hook* can be defined in each file after which the auto-generated code should be inserted. Configurable via environment variables. These control where app, enclave, EDL file, header and source code files are located. `$ sgx-lib/add_ocall.sh "void rewind([user_check] FILE* ↪ file);"`
`generate_error_codes.sh`[b]	✓	✓	Script that generates a helper function[c] to translate error codes into human-readable descriptions. Parses the SDK's `sgx_error.h` for error code descriptions.
`sgx_lib_t_¬stdio.h`[d]	✓		Proxies to input/output functions of the C library outside of the enclave. Part of the shim C library in Figure 6.1. Depending on the configuration, data is passed in plain text or transparently encrypted or sealed.
`sgx_lib_t_¬util.h`[e]	✓		Trusted helper functions. String formatting and return type checks. On errors, meaningful descriptions are printed (if transparent encryption/sealing is not activated).
`sgx_lib_t_debug.h`[f]	✓		Trusted helper functions for debugging. Log messages and `printf`.
`sgx_lib_t_¬crypto.h`[g]	✓		Wrapper functions for SDK encryption and sealing. Can calculate the sealed data size for a given plain text size (nonce plus a multiple of the block cipher size).
`sgx_lib_u_¬util.h`[h]		✓	Untrusted helper functions. Return type checks (see trusted util functions) and enclave setup/teardown.

Table 5.1.: Helper library overview. The important components (scripts, header files) are listed. The columns *trusted* and *untrusted* define where the code can be used (enclave or wrapper). The library includes two scripts to easily extend it with new C library proxies and error messages. Also it contains wrapper functions and pre-generated C library proxies. An example is given for the `add_ocall.sh` script.

[a] `https://github.com/ftes/sgx-lib/blob/thesis/add_ocall.sh`
[b] `https://github.com/ftes/sgx-lib/blob/thesis/sgx_lib_t/generate_error_codes.sh`
[c] `https://github.com/ftes/sgx-lib/blob/thesis/common/sgx_lib.c`
[d] `https://github.com/ftes/sgx-lib/blob/thesis/sgx_lib_t/include/sgx_lib_t_stdio.h`
[e] `https://github.com/ftes/sgx-lib/blob/thesis/sgx_lib_t/include/sgx_lib_t_util.h`
[f] `https://github.com/ftes/sgx-lib/blob/thesis/sgx_lib_t/include/sgx_lib_t_debug.h`
[g] `https://github.com/ftes/sgx-lib/blob/thesis/sgx_lib_t/include/sgx_lib_t_crypto.h`
[h] `https://github.com/ftes/sgx-lib/blob/thesis/sgx_lib_t/include/sgx_lib_t_util.h`

6. Related Work

The previous chapters already introduced a variety of related work. This chapter presents research more directly related to this thesis (e.g. with a similar problem definition).

Diverse solutions for hardening applications have been evaluated in existing research. Different avenues exist to approach the problem of hardening applications. Trusted computing is one alternative, cryptography – depending on the application – another.

This chapter first presents related research on secure databases. This section describes new application architectures and the use of cryptographic principles. Then research most related to this thesis is presented: application hardening using Intel SGX.

6.1. Hardened Databases

Database management software (DBMS) is standard software used by many applications. The data it operates on may be sensitive. The amount of data to process may also exceed viable enclave sizes. For these reasons, DBMS is deemed a good example that can be used in the case studies later on in this thesis.

Database software can be hardened with application-specific encryption in addition to standard solutions to trusted computing such as Intel SGX. Such approaches from research are now presented.[46] Hardened databases can be classified by their use of a secure location and the level of encryption homomorphism they employ. Table 6.1 shows the location of existing research prototypes in the design space spanned by these two dimensions.

The point in the design space that this thesis investigates is also indicated. The case studies in chapter 7 and chapter 8 further discuss this. The remainder of this section briefly elaborates on existing research. Excellent architecture diagrams for most of these solutions can be found in [3].

Arx[47] is the only DBMS in this list that does not perform some computation on encrypted data. It uses the trusted client's computer as a secure location. "Instead of embedding the computation into special encryption schemes [...], Arx embeds the computation into data structures, which it builds on top of traditional encryption schemes."

Arx uses an unmodified DBMS as foundation. On both the trusted location (the client) and the untrusted DBMS server a proxy is added. The client proxy rewrites queries and can decrypt results. Only it knows the decryption key. The client proxy also has to re-generate indices after usage.

[46] Commercial solutions are not presented. Examples are "Microsoft's Always Encrypted Service, currently deployed as part of SQL Server 2016, Skyhigh Networks, CipherCloud, Google's Encrypted Big Query, SAP's SEEED, Lincoln Labs."[47]

Secure location	Homomorphism of encryption scheme: None	Partial	Full
None			
Client	Arx[47]	CryptDB[48] Monomi[60]	
Co-processor		TrustedDB[7]	
FPGA		Cipherbase[2]	
SGX Enclave	*this thesis*		

Table 6.1.: Design space of hardened databases. Two dimensions are used for classification: secure location and homomorphic encryption. E.g. *TrustedDB* uses a co-processor and partially homomorphic encryption. Not all areas of the design space have been investigated. Promising uninvestigated areas are shaded green. This thesis explores secure databases using SGX enclaves and none-homomorphic encryption. Adding partially homomorphic encryption could benefit performance. Certain queries could be executed outside of the enclave (without decrypting the data in the enclave). Fully homomorphic encryption is still to inefficient but could enable complex queries an encrypted data. Based on [3].

Arx can – in a limited fashion – also securely evaluate confidential functions on the server. These functions must be expressed as garbled circuits, an implementation of two-party computation proposed by Yao.[64] Garbled circuits are used by Arx for range checks on the untrusted server. Arx's TCB consists of the client proxy, and potentially the entire software and hardware stack of the client if the proxy is not isolated.

CryptDB[48] uses "efficient SQL-aware encryption schemes". The data must be encrypted by the trusted client. The client must anticipate the expected query types and encrypt the data with matching encryption schemes. If unexpected query types are added later on, the client must re-encrypt parts of the data. CryptDB encrypts data with schemes that support DBMS operations such as equality checks, joining and searching. Such schemes have different characteristics: deterministic, order-preserving, partially homomorphic.

CryptDB uses "onion encryption" as an optimisation. Encrypted values are again encrypted with a different scheme. This minimises the required interaction of the client. If the server must perform a more complex query on a table (e.g. an equality join instead of just an equality select) the client provides a decryption key. With this key the server can peel off one more encryption layer of the onion. The encrypted value of the lower level is now encrypted with a scheme that supports the desired operation. Encryption schemes cannot be layered in any combination. For example, a deterministic encryption cannot be layered on top a non-deterministic one. It would not produce deterministic encryption of the original plain text.

CryptDB is implemented with a custom client proxy and user defined functions (UDFs) in a regular DBMS. CryptDB does not support the full SQL standard. The TCB is comparable to Arx.

Monomi[60] "builds on CryptDB's design of using specialised encryption schemes." In addition, Monomi splits the query execution into server and client parts. Monomi "executes as much of the query as is practical over encrypted data on the server, and executes the remaining components by shipping encrypted data to a trusted client, which decrypts data and processes queries normally."

Compared to CryptDB, Monomi is more flexible. By including the client in query execution more complex queries are possible. However query execution on the client is contrary to the idea of outsourcing computation. It also can require transfer of larger amounts of intermediate data. As an optimisation, Monomi proposes to pre-compute results for complex queries. Monomi's TCB is the comparable to Arx and CryptDB.

TrustedDB[7] is the first DBMS in this list to use trusted hardware on the server. TrustedDB actually runs two DBMS instances, one within the regular operating system and one on a secure co-processor. The trusted DBMS has a paging module that pulls in encrypted pages from the untrusted operating system when needed. The trusted DBMS knows the decryption key for the data.

A split query plan is generated, somewhat similar to Monomi. The query must be planned on the secure co-processor. As much computation as possible is performed on

encrypted data by the untrusted DBMS. The TCB includes the co-processor, and the trusted DBMS.

Cipherbase[2] also uses trusted hardware. A Field Programmable Gate Array (FPGA) is used to evaluate individual parts of the query. The FPGA is configured to run a stack machine and is not re-configured for every query.

Compared to TrustedDB, the software TCB on the server is smaller. The trusted hardware does not execute a full DBMS but only executes individual processing steps. However, Cipherbase also needs a trusted client to plan and optimise the queries. In TrustedDB, this functionality was provided by the trusted DBMS.

6.2. Hardening Applications with Intel SGX

This section presents related work on hardening applications using Intel SGX as a trusted computing solution. First, application-specific approaches are listed. Next, general approaches are described. The section concludes with a summary of the lessons learnt from this research.

Application-specific research focuses on hardening a specific application with SGX. The application in question may remain unmodified or be refactored. The approaches used in these papers can be abstracted and re-used to a certain degree:

Verifiable Confidential Cloud Computing (VC3)[53] was already described in section 4.5. Secure map-reduce jobs are executed in enclaves on an unmodified Hadoop. This solution is special because Hadoop takes programs (jobs) as input. It is sufficient to protect these jobs. The Hadoop engine runs outside of the enclave. VC3 manages to protect against a malicious Hadoop engine by protecting the integrity of the overall result using only the map-reduce jobs.

Though a highly interesting approach, this technique is not applicable in general. For the use-case of DBMS, UDFs could potentially be executed in enclaves like jobs in Hadoop.

SecureKeeper[12] was also presented in section 4.5. The approach followed in the paper is more generally applicable. ZooKeeper data is protected within in enclaves. To this end, parts of the ZooKeeper functionality are refactored. The authors favour a *tailored enclave* over an *application enclave*. The authors analyse memory access speeds in SGX and give recommendations on memory management in enclaves. These are identical to SCONE, which shares many of its authors with SecureKeeper.

General approaches deal with reusable approaches for isolating applications with SGX:

Haven[8] was already described in subsection 3.3.2 as a trusted computing solution for application-level isolation. Haven isolates unmodified legacy applications in an enclave. A library operating system is also included in the enclave to minimise the exploitable interface between the enclave and the untrusted world.

SCONE[5] connects SGX and Docker[47] containers. Alternative enclave designs are evaluated as shown in Figure 6.1. Option a) is Haven's approach of including a library operating system in the TCB. This keeps the interface between enclave and untrusted system extremely narrow (comparable to the interface between VM and hypervisor), but inflates the TCB. Option b) minimises the size of the TCB. The C library implementation lives outside of the enclave. This results in a large interface at the level of the C library interface. Option c) is the middle ground. The C library is lives inside the enclave, resulting in an enclave interface at the level of system calls. The authors choose option c), the middle ground, for their container implementation.

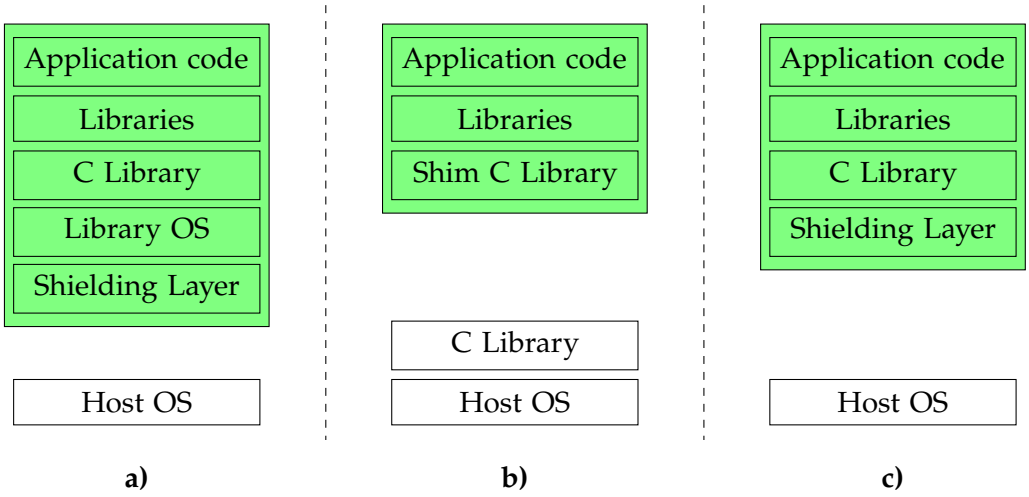


Figure 6.1.: Enclave design alternatives. The TCB contained by an enclave is shaded green. The components below that are untrusted. The options are: a) Library operating system inside the enclave. b) Minimal enclave size with an external C library. c) Untrusted system calls with an internal C library. Depending on the code included in the enclave, the TCB size and interface size vary. Reprinted from [5].

The performance of Intel's SDK is evaluated with regards to proxying system calls from inside the enclave to the host operating system. As it proves to be insufficient for handling many parallel system calls, the SDK is not used but replaced with a worker thread pool in the untrusted wrapper. The performance is evaluated based on different unmodified legacy applications. To measure the performance of the "file shield" (data sealing), SQLite is run in an enclave.

SCONE also evaluates how the process of authoring, provisioning and executing Docker images can be secured using attestation.

Software Partitioning case study[6] evaluates different approaches to partitioning OpenSSL into enclaves. Different partitioning schemes are identified, the most important of which are:

[47] `https://www.docker.com`

1. Separate enclaves by functionality. The enclave code must enforce isolation between different data sets within one enclave (e.g. between tenants).
2. Separate enclaves by data set (e.g. tenant). Related data lives in a single enclave which contains all functionality related to that data.

The two options can be combined. If non are used, the entire application lives in a single enclave (similar to Haven). If both are combined, there is an enclave per functionality per data set. Exploiting a single enclave reveals a minimal amount of data. However, higher decomposition requires more complex interaction. This may lead to new security issues.

The effect of separating enclaves by data set is questionable. If an exploit is found for an enclave, it can likely be applied to the enclaves of all tenants.

SCONE and the partitioning case study presented valuable design alternatives for enclaves. These alternatives are evaluated in the case studies later on in this thesis.

7. KISSDB Case Study

Two case studies were conducted to validate the gathered knowledge. The goal of both case studies is to harden an existing DBMS. Database software is chosen because it is a good example application for trusted computing. The data may be sensitive and require protection from the infrastructure provider and other tenants.

The first case study examines KISSDB[48], the "simplest key/value store you'll ever see, anywhere. It's written in plain vanilla C using only the standard string and FILE [input/output] functions."[49]. KISSDB stores key/value pairs of fixed size. It does not provide any processing, but only a put/get interface plus iterators. In this regard it is similar to ZooKeeper. Figure 7.1 shows the simplicity of KISSDB's database file layout.

In this case study, Intel SGX is used to protect the data KISSDB operates on. The code is not printed in this chapter or the appendix. Please refer to the Git repository.[50] The following aspects are *out of scope* for this case study.

- Attestation and secure communication channels.
- File integrity and freshness (replay attacks).
- Securely provisioning an encryption key.

The focus is on which part of KISSDB can be extracted, and transparently securing it with the helper libraries encryption/sealing features.

7.1. Design

This section discusses the design decisions for hardening KISSDB. The resulting architecture is shown in Figure 7.2.

Shim C library (see Figure 6.1). This option is the easiest to implement, but results in the largest enclave interface. This approach also incurs the performance overhead of the SDK-generated proxies. The ease of implementation outweighed the other two negative aspects.

Separate enclaves by data set. One enclave is set up per `open()` invocation. As KISSDB does not provide locking, only one enclave should be set up per database file.

Entire legacy code in enclave. KISSDB is not sub-divided into trusted and untrusted functionality. A single enclave is used for all trusted functionality. KISSDB is so small, it is

[48] Keep it Simple Stupid DataBase
[49] Original code: `https://github.com/adamierymenko/kissdb`
[50] Fork with SGX hardening:`https://github.com/ftes/kissdb-sgx`

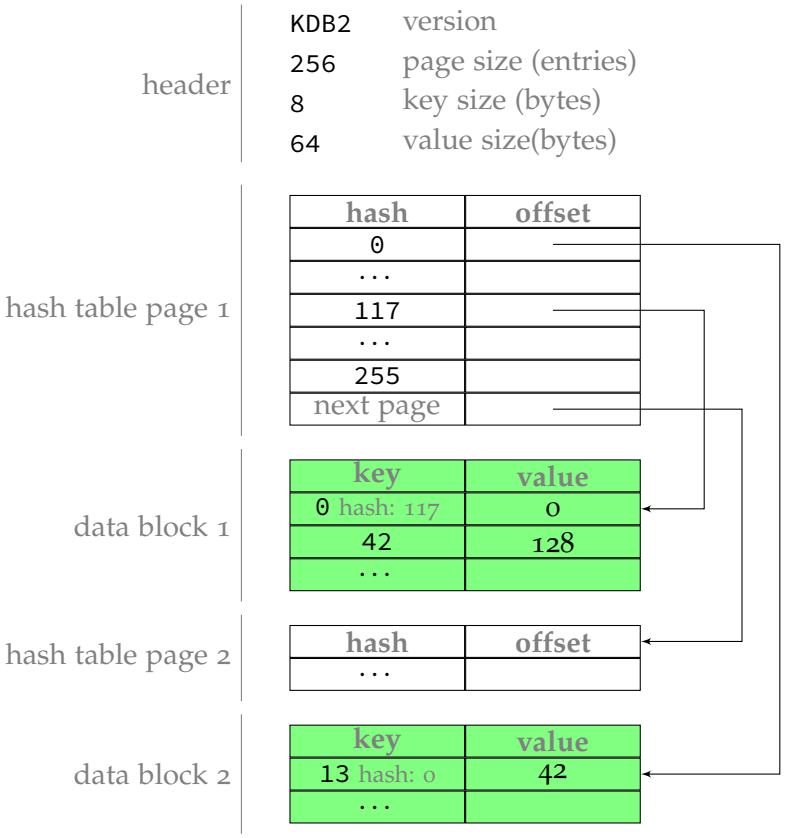


Figure 7.1.: KISSDB file layout. Implicit data structures (tables) are visualised. Text in grey is added as an explanation, but not present in the file. The destination of file offsets in the *offset* column are visualised as arrows. New data (key/value pair) is appended to the end of the file. A new hash table entry is inserted in the first page where the bucket is not yet occupied. A new hash table page is appended when all existing pages have an entry for the bucket in question. In the hardened version, data blocks are encrypted (shaded in green). Meta data (header and hash tables) is not encrypted.

difficult to identify a part that should be pulled out. KISSDB does not support any data processing, which otherwise would have been a likely candidate.

Plain text meta data. While at rest, data is encrypted. This means the payload is written to the database file in encrypted form (see Figure 7.1). The meta data (header, hash tables) are written as plain text. This keeps the required changes to the legacy code base to a minimum as discussed in section 7.2. This has the following security implications:

- The meta data is not protected. This includes the number of entries as well as the key and value size.
- The key hashes are not encrypted. If the hash scheme is not cryptographically secure, an attacker may learn information about the hash values.
- Also, if the key space is small or non-uniformly distributed, an attacker may learn information about the keys by pre-computing all (or all likely) key hashes.

The file content of original and hardened KISSDB files is compared in Appendix A.

Iterator outside of enclave. A KISSDB iterator is a cursor which allows iterating through all values. The cursor's position is identified by the hash table page number and item offset within that page. Several iterators can exist in parallel for a single database. The iterator is something that inherently belongs to the consumer using the iterator. The iterator data (page number and page offset) is held outside of the enclave. This way the enclave remains stateless. As the meta data is stored in plain, this is not an additional security risk.

Encrypt with custom key instead of sealing. Data sealing encrypts the data with a key derived from the enclave's identity. This identity is based only on the initial state (the loaded code). That means it is the same even if the enclave is initialised several times for different database files. Sealing the data would allow a consumer to read all other database files. Instead, the user has to specify the encryption key when creating the KISSDB instance (`open()` in Figure 7.2).

7.2. Implementation

This section highlights some implementation details.

Proxies in untrusted wrapper: The untrusted wrapper acts as a proxy to the enclave. The `open` and `close` functions must also set up and destroy the enclave. For this, they use the library's helper functions. Listing 7.1 shows parts of the enclave's interface definition.

Listing 7.1: kissdb.edl (extract).[51] Two E-calls from the trusted section of the EDL file. The `get` E-call shows how the `size` annotation is used to define the length of the parameters `key` and `value` through further parameters. The annotation `in` is used for the key, and `out` for the value. This tells the SDK to copy the key into the enclave before execution, and the value out of the enclave after execution.

[51] `https://github.com/ftes/kissdb-sgx/blob/thesis/kissdb_t/kissdb.edl#L7`

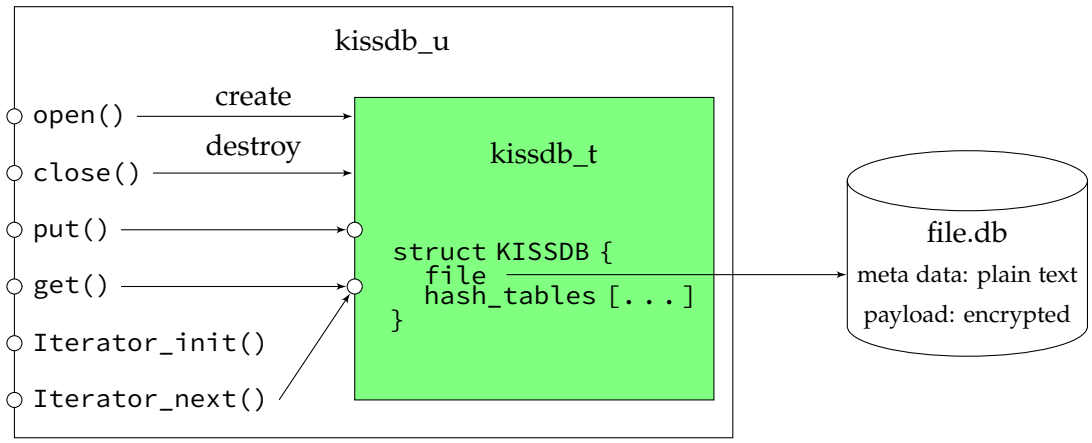


Figure 7.2.: Hardened KISSDB architecture. The legacy code is moved to an enclave (`kissdb_t`). The untrusted wrapper (`kissdb_u`) sets up the enclave and proxies `put`/`get` calls to the enclave. The shim C library transparently encrypts (or seals) the payload written to the file. The meta data is written as plain text. The hash tables are held in enclave memory for fast navigation.

```
7  public void KISSDB_close_ecall();
8  public int KISSDB_get_ecall([in, size=key_size] const void *key, [out, size=
       ↪ value_size] void *vbuf, unsigned long key_size, unsigned long value_size);
```

Listing 7.2 shows the corresponding implementation of the untrusted wrapper which delegates to the SDK-generated proxies.

Listing 7.2: kissdb_u_wrapper.c (extract).[52] Most functions in the wrapper simply delegate to the enclave through the SDK-generated E-call proxies. The `get` function adds the `_size` parameters, taken from the database structure. This informs the SDK which amount of memory to copy to/from the enclave for the `key` and `value` parameters. The `close` function most also tear down the enclave which is simplified through the helper library.

```
23 void KISSDB_close(KISSDB *db) {
24   // freeing memory and memsetting as performed by this ecall is not really
         ↪ necessary, as we are destroying enclave anyway
25   KISSDB_close_ecall(db->eid);
26
27   destroy_enclave(db->eid);
28   memset(db,0,sizeof(KISSDB));
29 }
30
31 int KISSDB_get(KISSDB *db, const void *key, void *vbuf) {
32   int retval;
33   KISSDB_get_ecall(db->eid, &retval, key, vbuf, db->key_size, db->value_size);
34   return retval;
35 }
```

Plain text meta data: For the case study, the library was configured to transparently encrypt all file input/output (see chapter 5). The meta data should be output as plain text, so

[52] https://github.com/ftes/kissdb-sgx/blob/thesis/kissdb_u/kissdb_u_wrapper.c#L23

a distinction has to be made between meta data and payload. This option requires the least changes to KISSDB's code.

The key and value size (which are also written to the file header) are adapted to include the cryptographic nonce and rounded to the next cipher block size. By keeping the header and hash tables in plain text, KISSDB's file navigation logic does not have to be altered. The offset calculation is preserved.

Listing 7.3 shows a *diff* command for the code changes to KISSDB.

Listing 7.3: Diff command to view changes to KISSDB's code. The hardened version was forked from the original at commit `37194e`.

```
1 cd kissdb_t
2 wget --no-check-certificate -O kissdb.c.orig https://raw.githubusercontent.com/
      ↪ adamierymenko/kissdb/37194e7019abfdcec95fe21d6cb2eb8debe78faf/kissdb.c
3 diff kissdb.c.orig kissdb.c
```

Listing 7.4 shows the most relevant parts of the diff.

Listing 7.4: kissdb.c diff (extract). The meta data is written and read as plain text using the `_insecure` library functions. The other file input/output operations (for payload) are transparently encrypted by the library. Also, the `encryption_key` is provided to the enclave during setup with the `open()` call.

```
12 @@ -42,7 +44,8 @@
13    int mode,
14    unsigned long hash_table_size,
15    unsigned long key_size,
16  - unsigned long value_size)
17  + unsigned long value_size,
18  + uint8_t *encryption_key)
19   {
20    uint64_t tmp;
21    uint8_t tmp2[4];
22 @@ -74,16 +77,17 @@
23    }
24    if (ftello(db->f) < KISSDB_HEADER_SIZE) {
25     /* write header if not already present */
26  +  /* header data is not sensitive -> unencrypted */
27     if ((hash_table_size)&&(key_size)&&(value_size)) {
28      if (fseeko(db->f,0,SEEK_SET)) { fclose(db->f); return KISSDB_ERROR_IO; }
29      tmp2[0] = 'K'; tmp2[1] = 'd'; tmp2[2] = 'B'; tmp2[3] = KISSDB_VERSION;
30  -   if (fwrite(tmp2,4,1,db->f) != 1) { fclose(db->f); return KISSDB_ERROR_IO; }
31  +   if (fwrite_insecure(tmp2,4,1,db->f) != 1) { fclose(db->f); return
      ↪ KISSDB_ERROR_IO; }
```

The file content of original and hardened KISSDB files is compared in Appendix A.

Different trusted/untrusted data structures: The database structure is used both inside and outside of the enclave. Different fields are required inside and outside of the enclave. The hash tables are held only in enclave memory to facilitate encrypting them in future.

The untrusted wrapper on the other hand must hold the enclave ID. This is needed to access E-calls and destroy the enclave. Listing 7.5 shows how the alternate structures are defined using macros.

Listing 7.5: kissdb.h diff (extract).

```
26   typedef struct {
27  -   unsigned long hash_table_size;
28      unsigned long key_size;
29      unsigned long value_size;
30  +
31  +#ifdef SGX_ENCLAVE
32  +   // hash tables live inside enclave only
33  +   unsigned long hash_table_size;
34      unsigned long hash_table_size_bytes;
35      unsigned long num_hash_tables;
36      uint64_t *hash_tables;
37      FILE *f;
38  -} KISSDB;
39  +#else
40  +   // identifies the enclave associated with this KISSDB instance in the
        ↪ untrusted application
41  +   uint64_t eid;
42  +#endif
43  +} DLLEXPORT KISSDB;
```

Include encryption key in interface: The KISSDB interface is extended to pass the database encryption key in the open call. This is the only modification to the external KISSDB interface and is shown in Listing 7.6.

Listing 7.6: kissdb.h diff (extract).

```
51  -extern int KISSDB_open(
52  +extern DLLEXPORT int KISSDB_open(
53      KISSDB *db,
54      const char *path,
55      int mode,
56      unsigned long hash_table_size,
57      unsigned long key_size,
58  -   unsigned long value_size);
59  +   unsigned long value_size,
60  +   uint8_t encryption_key[128]);
```

Passing the encryption key in the plain via the untrusted wrapper breaks security. This functionality was explicitly excluded from the scope of this case study for simplicity.

7.3. Open Issues

Several important aspects were excluded from the scope of this case study. These are open issues which break the security of the solution as it stands.

A list of these and other issues follows:

Attestation and key provisioning: In a production setting, the consumer should attest the enclaves identity and at the same time perform a key exchange with the enclave (see section 4.2). With the exchanged key, the database encryption key could securely be provisioned.

Ensure file integrity and freshness: Use cryptographic mechanisms to ensure file integrity. Include monotonic counters provided by the Intel SDK to ensure freshness of the file. If doing so, the possibility of migrating a database file between machines must be considered.

Cryptographic hash function: KISSDB uses the *djb2* hash function[53] to compute key hashes. This is not a cryptographic hash function. The hash tables (which are not encrypted) thus may leak information about the keys, even if the key space is large and uniformly distributed. It should be replaced with a cryptographic hash function.

Deterministic file layout: The file layout is deterministic. If values are added in the same order, the file layout is always the same. If the consumer's behaviour is known, this opens the door for known plain text attacks. This could occur if a consumer writes a fixed value upon first opening the database (e.g. version information).

7.4. Conclusion

The scope of the KISSDB study was limited in many regards. The case study was however successful in two aspects. Firstly, it helped validate the usefulness of the helper library. Secondly, the design alternatives for hardening applications gathered from related work (section 6.2) could be applied.

[53] `http://www.cse.yorku.ca/~oz/hash.html`

8. SQLite Case Study

The second case study examines hardening SQLite, "SQLite is a self-contained, high-reliability, embedded, full-featured, public-domain, SQL database engine. SQLite is the most used database engine in the world."[54] The results of this case study are limited to concept work without any implementation.

The architecture and design decisions of SQLite are described in [28]. They are only briefly explained in this chapter. Please refer to this book for further details.

SQLite was chosen for the following reasons:

Comparably small: Compared to KISSDB, SQLite is a production-level DBMS. In the world of production-level DBMS however, SQLite is a comparably small piece of software. It is not a stand alone server application but rather an embedded DBMS. "With all features enabled, the library size can be less than 500KiB".

Modularised: As described in [28, ch. 2.6], SQLite has a very modular architecture. Figure 8.1 shows the modules and a potential enclave border.

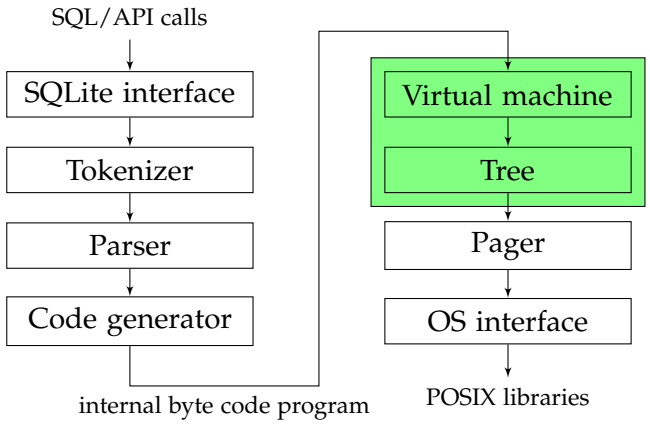


Figure 8.1.: SQLite architecture. SQLite is divided into a front end and back end. The front end translates incoming SQL statements (and SQLite API calls) into an internal byte code program. The byte code is executed by the virtual machine – also called Virtual Database Engine (VDBE). The VDBE is part of the back end, and operates on data through the tree module. The VDBE and tree (shaded green) process the data and hold data and derived structures in memory. These two modules are a prime candidate for enclave protection. Reprinted from [28].

[54] https://www.sqlite.org/

The Virtual Database Engine (VDBE) – as the virtual machine is also called – is at the core of SQLite. The front end compiles a program for the VDBE, and the rest of the back end is the data source.

8.1. Analysis

In order to find potential partitioning strategies for hardening SQLite, the run time behaviour was analysed. Valgrind[55] was used to record calls on function level. KCachegrind[56] was used to analyse, filter and export Valgrind's output. Listing 8.1 shows the steps.

Listing 8.1: Record SQLite call graph. Valgrind and KCachegrind are used in combination to first record and then analyse the call graph. An `insert` SQL statement is executed on a new database through the SQLite command line interface.

```
1  # Run Valgrind to record SQLite's calls
2  $ valgrind --tool=callgrind ./sqlite3
3
4  # Execute SQLite instructions (create new database, insert and select).
5  sqlite3$ .open test.db
6  sqlite3$ CREATE TABLE tbl ( value TINYINT );
7  sqlite3$ INSERT INTO tbl VALUES ( 42 );
8  sqlite3$ SELECT * FROM tbl;
9
10 # Exit Using Ctrl+d
11 Ctrl+d
12
13 # Execute KCachegrind to analyze output
14 $ kcachegrind callgrind.out.<id>
```

Figure 8.2 shows the call graph after some additional post processing steps. Most notably, the call graph nodes were shaded based on which SQLite module they belong to. The functions (nodes) were mapped to their source or header file via a script. The SQLite source files were manually mapped to the SQLite modules.

Not all files could be attributed to exactly one module. Nodes shaded grey can not clearly be attributed to a module. The complete call graph is far too large to visualise. Figure 8.2 contains only nodes that incur at least one percent of the total cost as defined by Valgrind.[57]
A further visualisation of the same call graph in Appendix B is filtered not by cost, but by depth. This is even larger, but can convey an overview of the module interaction through the coloured shading.

[55] http://valgrind.org/
[56] http://kcachegrind.sourceforge.net/html/Home.html
[57] http://valgrind.org/docs/manual/cl-manual.html#cl-manual.functionality

8.2. Concepts

Based on the related work and SQLite run time analysis, two different hardening concepts are proposed. They are not implemented due to time constraints.

1. **Extract the VDBE and tree module into an enclave**. This approach modifies the SQLite code and extracts the security critical part. This separates the enclave by functionality (section 6.2). Figure 8.1 shows the proposed boundaries of the enclave: it should include the VDBE and tree module.

   The premise is that it is sufficient and secure to protect the VDBE and the tree module. The enclave would provide an interface byte code level. The caller would pass a byte code program in an E-call. An in-depth analysis of the data flow is necessary to judge the security implications of this separation.

   The VDBE executes the compiled byte code program. It fetches entries from the tree module, which in turn access the disk via the pager. The tree module is a good lower boundary (in terms of the position in the architecture diagram), because it accesses the disk at the level of pages. The tree module could be adapted to write and read encrypted values, but pass the plain text on to the VDBE.

   The VDBE then performs the actual processing, based on the byte code program. It yields individual results row by row to the caller. The output has to be encrypted row by row if the interface should remain the same. This leaks the number of result rows for every call. The VDBE is a good upper boundary, because this keeps the entire front end out of the TCB. This is significant, as the code generator alone contains 40 percent of the entire SQLite code.[28] The front end is not involved in data processing, but only responsible for byte code generation.

   If the byte code is generated by an untrusted front end, it must be ensured that the byte code is not harmful. Also, the output of the VDBE should not leak any information. It should be encrypted to a secure channel established during attestation.

   In reality, the modules are not so well separated as the high-level architecture diagram would have readers believe. This can be seen in the call graph (Figure 8.2). Especially calls back and forth between front end and back end are problematic for extracting an enclave. While enclaves support O-calls and E-calls, the data that must be passed back and forth is the root of the problem. If the untrusted front end must operate on the data it must be passed as plain text, which may break security.

2. **User-defined functions (UDFs) in enclaves**. This solution leaves the KISSDB code base untouched. Instead, sensitive data is processed in enclaves via UDFs.[58] Figure 8.3 shows the architecture of this solution.

   The approach is very similar to VC3.[53] It uses separate processing components (UDFs here, jobs in VC3) which can be plugged into the main processing engine. The approach separates enclaves by functionality. There is one enclave per UDF.

[58] Registered via `create_function`: `http://www.sqlite.org/c3ref/create_function.html`

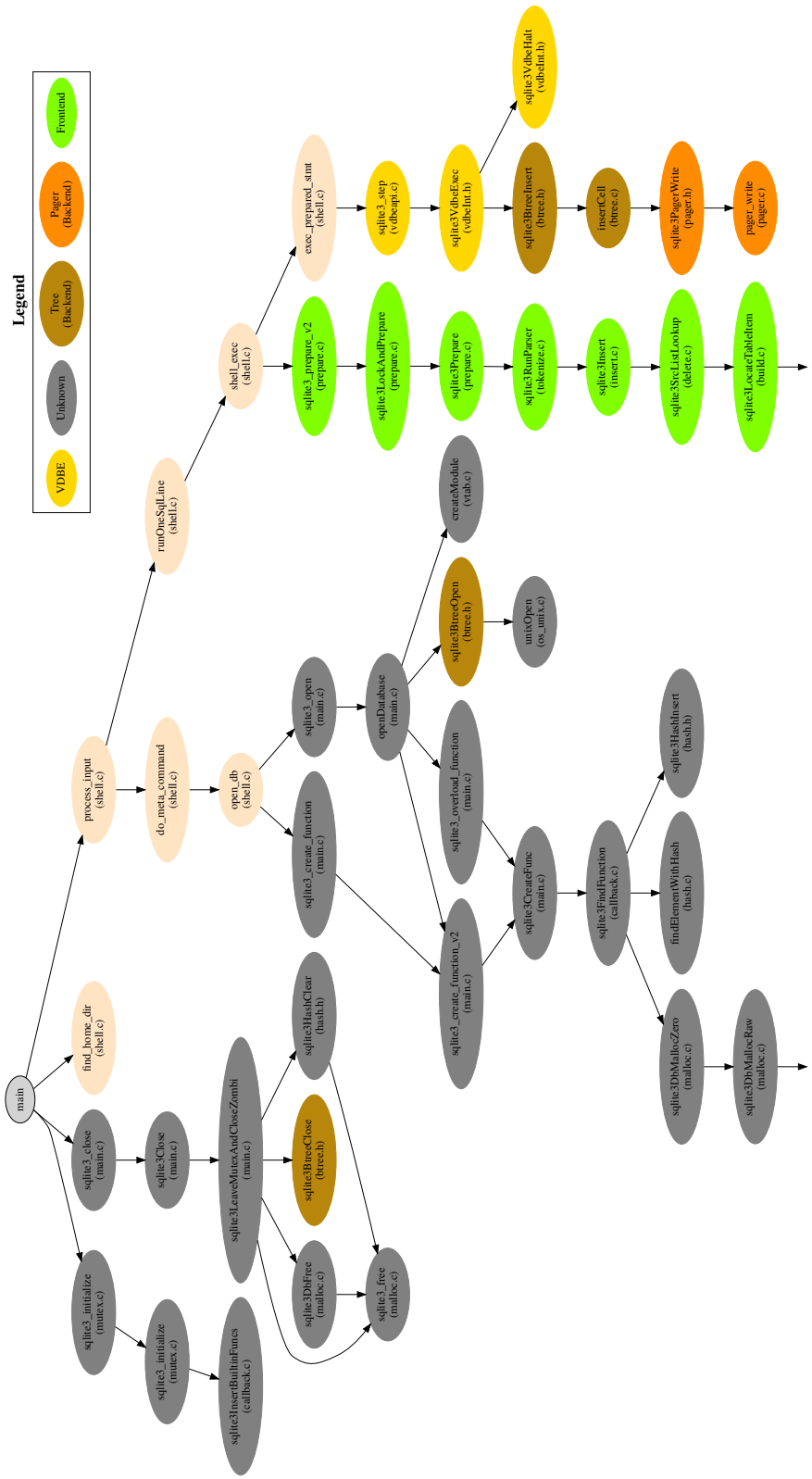


Figure 8.2.: SQLite call graph for insert statement (top half).

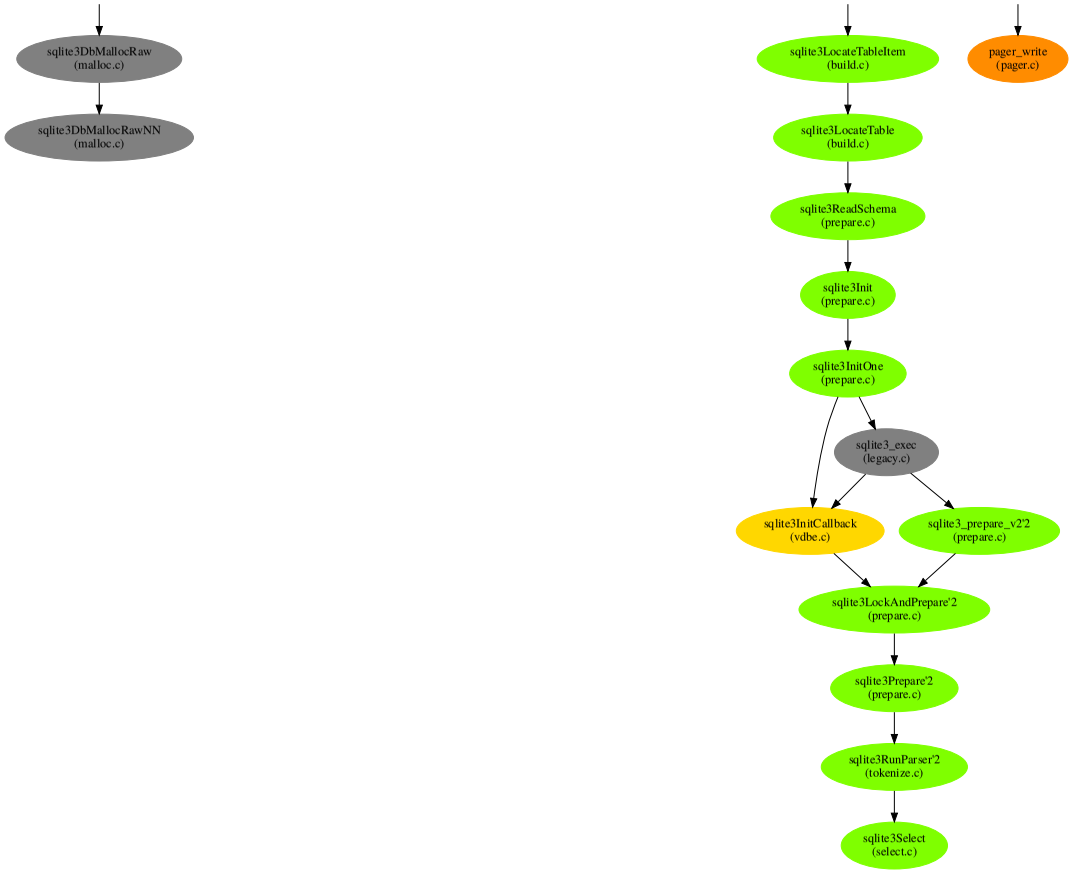


Figure 8.2.: SQLite call graph for insert statement (bottom half). Only function calls (nodes) with a Valgrind cost greater than 1 percent are shown. Nodes are shaded according to the SQLite module they belong to (see the legend). The modules are distinguished by the header or code file the function is defined in. The file is given in brackets.

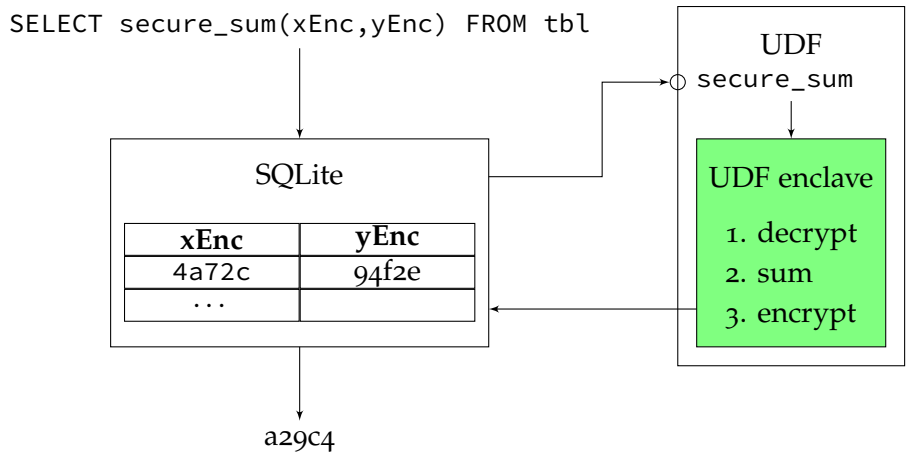


Figure 8.3.: SQLite hardening with user-defined functions (UDFs). A function is registered with a SQLite connection using `sqlite3_create_function`. The registered function is an untrusted proxy that delegates the call to the enclave. The enclave has been provisioned with the encryption key for the data. It decrypts the operands and encrypts the result before passing it back. SQLite itself handles only binary encrypted data.

Compared to VDBE and tree extraction, the advantages of the UDF approach are as follows:

- This approach is far easier to implement. SQLite must not be modified. This facilitates compatibility with future versions.
- The security guaranteed by this approach is easier to reason about, as the data flow is very clear.

There are also drawbacks:

- Functions have to be re-implemented. SQLite has optimised processing implementations. They work well with the tree and pager module. None of these existing operations can be used on encrypted data (for non-homomorphic encryption).
- UDFs can only operate at row level. Relational operations such as joins will not work if non-deterministic encryption schemes are used. Aggregations can be defined as UDFs (with an interface similar to reducers in functional programming). A combination with the techniques used by other hardened databases in section 6.1 is possible (e.g. onion encryption).
- Performing many E-calls is inefficient. Even if the Intel SDK is not used, E-calls still have a performance overhead. When processing a query that touches many rows, the enclave will be called many times. The resulting context switches into the enclave degrade performance.
- Information leakage at the field level. The enclave returns encrypted values. However, it is increasingly likely that some information is leaked. If the value space of a field is small or non-uniformly distributed, information may be learned even

from an encrypted value (take a binary field with deterministic encryption as an extreme example).

8.3. SQLite in the Intel SGX SDK

The Intel SGX SDK for Linux, which is open source, also includes SQLite. The way in which it is used by enclaves is different to the presented concepts and the related work. For the specific use case of the SDK, it is a simple and interesting alternative.

Multiple references to SQLite can be found in the SDK's code.[59]. Browsing the code, it seems that SQLite is used to provide monotonic counters.[60] The comments in the repository often mention the CSME, which stands for Converged Security and Manageability Engine. Monotonic counters are provided in hardware by the manageability engine.[32] They are a limited resource. SQLite is apparently used to multiplex the hardware monotonic counters into several virtual counters. This is a feature provided by Intel's platform software (PSW).

The SQLite database is stored outside of the enclaves. For the counters, it is sufficient to integrity protect the database. The code indicates that the data is stored in tree form. If stored as a Merkle tree, it is sufficient to securely store the hash of the root node in the enclave. The integrity of the entire tree can be verified from this root hash.

This approach leaves the payload visible as plain text. For the use case, integrity and replay protection are sufficient. This allows for a far simpler solution than the concepts proposed in this case study. The SDK approach however falls short of providing protected processing of data, which is the goal of this thesis.

8.4. Conclusion

In this case study, two concepts to hardening SQLite were proposed. These were based on a brief run-time analysis of SQLite's module interaction. Also, related work provided an inspiration for the UDF approach. Due to time constraints, neither of the approaches could be evaluated in-depth or implemented. The unrelated approach of the SGX SDK was also presented as an example of storing integrity-protected data outside enclaves.

[59] https://github.com/01org/linux-sgx/search?q=sqlite

[60] https://github.com/01org/linux-sgx/blob/1115c195cd60d5ab2b80c12d07e21663e5aa8030/psw/ae/pse/pse_op/monotonic_counter_database_sqlite_rpdb.cpp

9. Conclusion

This thesis showed how applications can best be hardened with the technology that is available today. In a first step in chapter 2, trusted computing was identified as the correct approach for the type of hardening in mind. Alternative approaches based on cryptography alone are either too limited (garbled circuits) or not yet practical (fully homomorphic encryption, encrypted CPU).

A major contribution of this thesis is the survey and systematic comparison of trusted computing solutions in chapter 3. Intel SGX was identified as the best trusted computing technology for hardening applications. The isolated TCB is kept small and the CPU's full processing power can be used. Developers can focus on their application and do not have to provide their own trusted computing infrastructure as on e.g. TrustZone.

The thesis provided a high-level overview of SGX and summarised criticism and security issues from research in chapter 4. A helper library for Intel's SDK was developed (chapter 5) and made available for public use.

Architectural design patterns were extracted from related work in chapter 6. These can be re-used in future work. The case studies (chapter 7, chapter 8) provide a step-by-step template for application hardening. The applied reasoning and helper library should prove useful to developers targeting a similar problem. Two concepts were derived for the second case study. Implementing one of these approaches is still open work. Also, attestation with SGX still has to be explored. Only with remote attestation does trusted computing unfold its full potential.

This thesis dealt with hardening legacy applications. Intel SGX provides the foundation for entirely new and innovative applications not possible without trusted computing. This line of research has vast potential.

Intel has made a serious investment in developing SGX. The success of SGX is still far from decided – too much is still unclear. Intel has set itself up in a good position in case SGX succeeds. But judging from the criticism being voiced and the limited amount of applications it may have pushed it's luck too far. Further commercial solutions comparable to SGX can be expected to be developed by Intel's competition. Once that time comes, research comparing these solutions will be of interest. Apart from the security aspects and development model, the factor that decides over the winning solution might well be the business model.

A. KISSDB Database Files

Listing A.1: Plain KISSDB database file. The file is shown in hex editor view, with the binary content on the left, and the ASCII characters on the right. Both meta data and payload are in plain text.

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  4B 64 42 02 00 04 00 00 00 00 00 00 08 00 00 00  KdB.............  // header:
    ↪ KDB2, hash table size (0004...), key size (08...) = 8 byte
00000010  00 00 00 00 40 00 00 00 00 00 00 00 0C 14 05 00  ....@...........  // header:
    ↪ value size (40...) = 64 byte
00000020  00 00 00 00 5C 0A 07 00 00 00 00 00 4C 4C 00 00  ....\.......LL..  // BEGIN
    ↪ first hash table page (incl.  0C 14 05 00  in previous line)
00000030  00 00 00 00 CC 3F 01 00 00 00 00 00 4C F6 03 00  ....I?......Lo..
00000040  00 00 00 00 9C EC 05 00 00 00 00 00 EC E2 07 00  ....oei......ia..
...
00001030  00 00 00 00 7C 7B 06 00 00 00 00 00 DC 5E 00 00  ....|{......U^..
00001040  00 00 00 00 24 20 00 00 00 00 00 00 64 47 03 00  ....$ ......dG..  // hash
    ↪ table entry for item with key  0  is  24  20 , at offset  1044  (as expected)
00001050  00 00 00 00 BC 5D 05 00 00 00 00 00 0C 54 07 00  .... 1/4 ].......T..
...
00001FF0  00 00 00 00 7C 3B 04 00 00 00 00 00 CC 31 06 00  ....|;......I1..
00002000  00 00 00 00 E4 65 00 00 00 00 00 00 2C 27 00 00  ....ae......,'..
00002010  00 00 00 00 AC DD 02 00 00 00 00 00 DC 70 00 00  ....!Y......Up..  // END
    ↪ first hash table page, last entry is offset of next hash table page  DC 70  ->
    ↪  70 DC  (little endian)
00002020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................  // BEGIN
    ↪ first entry at 2024: key (0), value ([0,0,0,0,0,0,0,0])
00002030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................  // entry is
    ↪  72byte (0x48 hex) long, next entry starts at  0x2024 + 0x48 = 0x206C
00002040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................  // 72 byte:
    ↪  9 64-bit integers (1 key, 8 value) ->  9 * 64 / 8 = 72byte
00002050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00002060  00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00  ................  // BEGIN
    ↪ second entry
00002070  00 00 00 00 01 00 00 00 00 00 00 00 01 00 00 00  ................
00002080  00 00 00 00 01 00 00 00 00 00 00 00 01 00 00 00  ................
00002090  00 00 00 00 01 00 00 00 00 00 00 00 01 00 00 00  ................
000020A0  00 00 00 00 01 00 00 00 00 00 00 00 01 00 00 00  ................
000020B0  00 00 00 00 02 00 00 00 00 00 00 00 02 00 00 00  ................  // BEGIN
    ↪ third entry
...
000070C0  00 00 00 00 1E 01 00 00 00 00 00 00 1E 01 00 00  ................
000070D0  00 00 00 00 1E 01 00 00 00 00 00 00 C4 64 05 00  ............Ad..  // BEGIN
    ↪ second hash table page at offset 70DC (as linked from first page)
000070E0  00 00 00 00 14 5B 07 00 00 00 00 00 0C BD 00 00  .....[....... 1/2 ..
```

```
30  000070F0  00 00 00 00 8C B0 01 00 00 00 00 00 04 47 04 00  ....OE?.......G..
31  ...
```

Listing A.2: Hardened KISSDB database file. The file is shown in hex editor view, with the binary content on the left, and the ASCII characters on the right. The payload is encrypted, while the meta data is in plain text.

```
1  Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
2
3  00000000  4B 64 42 02 00 04 00 00 00 00 00 00 08 00 00 00  KdB.............  // header:
       ↪ identical (unencrypted)
4  00000010  00 00 00 00 40 00 00 00 00 00 00 00 7C B6 07 00  ....@.......|?..
5  00000020  00 00 00 00 AC FB 0A 00 00 00 00 00 BC 69 00 00  ....!u...... 1/4 i..
6    // BEGIN first hash table page (unencrypted)
7  00000030  00 00 00 00 DC D4 01 00 00 00 00 00 3C DA 05 00  ....UO......<U..
8  00000040  00 00 00 00 6C 1F 09 00 00 00 00 00 9C 64 0C 00  ....l.......oed..
9  ...
10 00001030  00 00 00 00 8C 0D 0A 00 00 00 00 00 AC 88 00 00  ....OE.......!^..
11 00001040  00 00 00 00 24 20 00 00 00 00 00 00 14 CC 04 00  ....$ .......I..
12 // hash table entry for first inserted item has not changed (offset remains the same):
       ↪ 24 20
13 00001050  00 00 00 00 4C 31 08 00 00 00 00 00 7C 76 0B 00  ....L1......|v..
14 ...
15 00001FF0  00 00 00 00 8C 4D 06 00 00 00 00 00 BC 92 09 00  ....OEM...... 1/4 '..
16 00002000  00 00 00 00 64 94 00 00 00 00 00 00 DC 2B 00 00  ....d"......U+..
17 00002010  00 00 00 00 3C 31 04 00 00 00 00 00 AC A6 00 00  ....<1......!|..
18 // END first hash table page
19 00002020  00 00 00 00 7E 6E DF 1F 2C 34 F5 4E BD CD D4 66  ....~nss.,4oN 1/2 IOf
20 // BEGIN first entry at 2024: ctr_nonce=~nss.,4oN, data= 1/2 IOf...
21 00002030  D4 53 B7 C4 8A 74 FB 5A 18 67 71 65 1B 80 A9 AD  OS.AStuZ.gqe.EUR(C).
22   // encrypted key (incl. ctr_nonce) is 16byte (0x10 hex) long, encrypted value starts
       ↪ at  0x2024 + 0x10 = 0x2034
23 00002040  35 AF 88 7B 11 A6 E0 C4 A5 58 3C 61 4C EF FF DE  5?^{.|aAJPYX<aLiyTH
24 ...
```

B. SQLite Call Graphs

Figure B.1.: SQLite call graph for insert statement. The depth is limited to a maximum of 15 callees (descendants). The call graph is too large for detailed analysis. Rather, the shading should convey an intuition of how the modules are interleaved. See Figure 8.2 for the colour legend. Nodes are coloured according to the SQLite module they belong to (see the legend). The modules are distinguished by the header or code file the function is defined in. The file is given in brackets.

Bibliography

[1] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. „Innovative technology for CPU based attestation and sealing“. In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. 2013, page 10. DOI: https://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing.

[2] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. „Orthogonal Security with Cipherbase“. In: *CIDR*. 2013. URL: http://research.microsoft.com/pubs/179425/cipherbase.pdf.

[3] Arvind Arasu, Ken Eguro, Raghav Kaushik, and Ravi Ramamurthy. „Querying Encrypted Data (Tutorial)“. In: 2013. URL: https://www.microsoft.com/en-us/research/publication/querying-encrypted-data-tutorial/.

[4] ARM. *ARM Security Technology — Building a Secure System using TrustZone Technology*. Technical report. PRD29-GENC-009492C. URL: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[5] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L Stillwell, et al. „SCONE: Secure linux containers with Intel SGX“. In: *12th USENIX Symp. Operating Systems Design and Implementation*. 2016.

[6] Ahmad Atamli-Reineh and Andrew Martin. „Securing application with software partitioning: A case study using SGX“. In: *International Conference on Security and Privacy in Communication Systems*. Springer. 2015, pages 605–621.

[7] S. Bajaj and R. Sion. „TrustedDB: A Trusted Hardware-Based Database with Privacy and Data Confidentiality“. In: *IEEE Transactions on Knowledge and Data Engineering* 26.3 (2014), pages 752–765. ISSN: 1041-4347. DOI: 10.1109/TKDE.2013.38.

[8] Andrew Baumann, Marcus Peinado, and Galen Hunt. „Shielding applications from an untrusted cloud with haven“. In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014. DOI: 10.1145/2799647.

[9] Michael Brenner, Henning Perl, and Matthew Smith. „How practical is homomorphically encrypted program execution? An implementation and performance evaluation“. In: *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*. IEEE. 2012, pages 375–382. DOI: 10.1109/TrustCom.2012.174.

[10] Michael Brenner, Jan Wiebelitz, Gabriele von Voigt, and Matthew Smith. „A smart-gentry based software system for secret program execution“. In: *Security and Cryptography (SECRYPT), 2011 Proceedings of the International Conference on*. IEEE. 2011, pages 238–244. URL: https://www.dcsec.uni-hannover.de/uploads/tx_tkpublikationen/SECRYPT_2011_31_CR.pdf.

[11] Michael Brenner, Jan Wiebelitz, Gabriele Von Voigt, and Matthew Smith. „Secret program execution in the cloud applying homomorphic encryption“. In: *Digital Ecosystems and Technologies Conference (DEST), 2011 Proceedings of the 5th IEEE International Conference on*. IEEE. 2011, pages 114–119. DOI: 10.1109/DEST.2011.5936608.

[12] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. „SecureKeeper: Confidential ZooKeeper using Intel SGX“. In: *Proceedings of the 16th Annual Middleware Conference (Middleware)*. 2016.

[13] Lily Chen, Joshua Franklin, and Andrew Regenscheid. *Guidelines on Hardware-Rooted Security in Mobile Devices (Draft). Methods and Techniques*. Technical report 800-38A. 2012. DOI: 10.6028/NIST.SP.800-38A. URL: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf.

[14] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. „Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems“. In: *SIGOPS Oper. Syst. Rev.* 42.2 (2008), pages 2–13. ISSN: 0163-5980. DOI: 10.1145/1353535.1346284.

[15] Victor Costan and Srinivas Devadas. „Intel SGX Explained.“ In: *IACR Cryptology ePrint Archive* (2016), page 86.

[16] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. „Sanctum: Minimal Hardware Extensions for Strong Software Isolation.“ In: *USENIX Security Symposium*. 2016, pages 857–874.

[17] Cyberlink. *Was sind die Mindestsystemvoraussetzungen für die Ultra HD Blu-ray-Filmwiedergabe?* July 12, 2017. URL: http://de.cyberlink.com/support/faq-content.do?id=19147 (visited on July 12, 2017).

[18] DoD Computer Security Center. *Trusted Computer System Evaluation Criteria*. DoD 5200.28-STD. Department of Defense. 1985. URL: http://csrc.nist.gov/publications/history/dod85.pdf.

[19] Jan-Erik Ekberg, Kari Kostiainen, and N. Asokan. „Trusted Execution Environments on Mobile Devices“. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. CCS '13. Berlin, Germany: ACM, 2013, pages 1497–1498. ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516758. URL: http://www.cs.helsinki.fi/group/secures/CCS-tutorial/tutorial-slides.pdf.

[20] Taher ElGamal. „A public key cryptosystem and a signature scheme based on discrete logarithms“. In: *IEEE transactions on information theory* 31.4 (1985), pages 469–472.

[21] Anand S. Gajparia and Chris J. Mitchell. „Trusted computing“. In: Springer. 2005.

[22] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. „Terra: A virtual machine-based platform for trusted computing“. In: *ACM SIGOPS Operating Systems Review*. Volume 37. 5. ACM. 2003, pages 193–206. DOI: 10.1145/1165389.945464.

[23] Rosario Gennaro, Craig Gentry, and Bryan Parno. „Non-interactive verifiable computing: Outsourcing computation to untrusted workers“. In: *Annual Cryptology Conference*. Springer. 2010, pages 465–482.

[24] Craig Gentry et al. „Fully homomorphic encryption using ideal lattices.“ In: *STOC*. Volume 9. 2009, pages 169–178.

[25] GlobalPlatform Inc. *TEE System Architecture*. Technical report. Version 1.1. 2017. URL: https://www.globalplatform.org/specificationsdevice.asp.

[26] Shay Gueron. „A Memory Encryption Engine Suitable for General Purpose Processors.“ In: *IACR Cryptology ePrint Archive* (2016), page 204.

[27] Debayan Gupta, Benjamin Mood, Joan Feigenbaum, Kevin Butler, and Patrick Traynor. „Using Intel Software Guard Extensions for Efficient Two-Party Secure Function Evaluation“. In: *Financial Cryptography and Data Security*. 2016.

[28] Sibsankar Haldar. „SQLite Database System Design and Implementation“. In: *Sibsankar Haldar* (2015).

[29] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. „InkTag: Secure Applications on an Untrusted Operating System“. In: *SIGPLAN Not.* 48.4 (2013), pages 265–278. ISSN: 0362-1340. DOI: 10.1145/2499368.2451146.

[30] Intel. *Intel Sawtooth*. URL: https://intelledger.github.io/introduction.html#proof-of-elapsed-time-poet (visited on July 12, 2017).

[31] Intel. *Intel SGX Developer Guide*. URL: https://01.org/intel-software-guard-extensions/documentation/intel-sgx-developer-guide (visited on July 17, 2017).

[32] Intel. *Intel SGX Evaluation SDK User's Guide for Windows OS. Developer Reference*. Version 1.1. 2015. URL: https://software.intel.com/sites/default/files/managed/33/70/intel-sgx-developer-guide.pdf (visited on Aug. 17, 2017).

[33] Intel. *Intel Software Guards Extensions Programming Reference*. Version 329298-002US. 2014. URL: https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf (visited on July 17, 2017).

[34] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. *Intel Software Guard Extensions: EPID Provisioning and Attestation Services*. Technical report. Intel, 2016. URL: https://software.intel.com/sites/default/files/managed/ac/40/2016%20WW10%20sgx%20provisioning%20and%20attesatation%20final.pdf (visited on July 17, 2017).

[35] Dennis Kafura. *Trusted Platform Module — Integrity Measurement, Reporting, and Evaluation*. URL: http://courses.cs.vt.edu/~cs5204/fall10-kafura-BB/Papers/Overheads/TPM.pptx (visited on Jan. 28, 2016).

[36] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. „Verena: End-to-end integrity protection for web applications". In: *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE. 2016, pages 895–913.

[37] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B Lee. „NoHype: virtualized cloud infrastructure without the virtualization". In: *ACM SIGARCH Computer Architecture News*. Volume 38. 3. ACM. 2010, pages 350–361. DOI: 10.1145/1816038.1816010.

[38] Sunny King. „Primecoin: Cryptocurrency with prime number proof-of-work". In: *July 7th* (2013).

[39] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. „MiniBox: A Two-Way Sandbox for x86 Native Code". In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, 2014, pages 409–420. ISBN: 978-1-931971-10-2. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_yanlin.

[40] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. „TrustVisor: Efficient TCB Reduction and Attestation". In: *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pages 143–158. ISBN: 978-0-7695-4035-1. DOI: 10.1109/SP.2010.17.

[41] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. „Flicker: An execution infrastructure for TCB minimization". In: *ACM SIGOPS Operating Systems Review*. Volume 42. 4. ACM. 2008, pages 315–328. DOI: 10.1145/1357010.1352625.

[42] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday 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*. ACM. 2013, pages 1–1. DOI: 10.1145/2487726.2488368.

[43] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. 2008.

[44] Emmanuel Owusu, Jorge Guajardo, Jonathan McCune, Jim Newsome, Adrian Perrig, and Amit Vasudevan. „OASIS: On achieving a sanctuary for integrity and secrecy on untrusted platforms". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM. 2013, pages 13–24. DOI: 10.1145/2508859.2516678.

[45] Pascal Paillier. „Public-key cryptosystems based on composite degree residuosity classes". In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 1999, pages 223–238.

[46] Ronald Perez, Reiner Sailer, Leendert van Doorn, et al. „vTPM: virtualizing the trusted platform module“. In: *Proc. 15th Conf. on USENIX Security Symposium*. 2006, pages 305–320. URL: https://www.usenix.org/legacy/event/sec06/tech/full_papers/berger/berger.pdf.

[47] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. „Arx: A Strongly Encrypted Database System.“ In: *IACR Cryptology ePrint Archive* (2016), page 591.

[48] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. „CryptDB: protecting confidentiality with encrypted query processing“. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. 2011, pages 85–100. DOI: 10.1145/2043556.2043566.

[49] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, et al. *fTPM: A Firmware-based TPM 2.0 Implementation*. Technical report. 2015. URL: http://research.microsoft.com/pubs/258236/msr-tr-2015-84.pdf.

[50] Ltd. Samsung Electronics Company. *White Paper: An Overview of the Samsung Knox Platform*. Technical report. 2016. URL: https://www.samsungknox.com/de/knox-technology/white-papers (visited on July 17, 2017).

[51] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. „Using ARM TrustZone to build a trusted language runtime for mobile applications“. In: *ACM SIGARCH Computer Architecture News*. Volume 42. 1. ACM. 2014, pages 67–80. DOI: 10.1145/2654822.2541949.

[52] Klaus Schmeh. „Enklavenhaltung. Die Trusted-Computing-Technologie SGX von Intel“. In: *iX* 2 (2017), pages 80–84. URL: https://heise.de/-3603810.

[53] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. „VC3: Trustworthy data analytics in the cloud using SGX“. In: *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE. 2015, pages 38–54. DOI: 10.1109/SP.2015.10.

[54] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. „Malware guard extension: Using SGX to conceal cache attacks“. In: *arXiv preprint arXiv:1702.08719* (2017).

[55] Nigel P Smart and Frederik Vercauteren. „Fully homomorphic encryption with relatively small key and ciphertext sizes“. In: *International Workshop on Public Key Cryptography*. Springer. 2010, pages 420–443.

[56] Udo Steinberg and Bernhard Kauer. „NOVA: a microhypervisor-based secure virtualization architecture“. In: *Proceedings of the 5th European conference on Computer systems*. ACM. 2010, pages 209–222. DOI: 10.1145/1755913.1755935.

[57] Raoul Strackx and Frank Piessens. „Fides: Selectively hardening software application components against kernel-level or process-level malware“. In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, pages 2–13. DOI: 10.1145/2382196.2382200.

[58] Trusted Computing Group. *TPM 1.2 Main — Part 1: Design Principles*. Technical report. Revision 116. 2011. URL: https://trustedcomputinggroup.org/tpm-main-specification/ (visited on July 17, 2017).

[59] Trusted Computing Group. *Trusted Platform Module 2.0 Library — Part 1: Architecture*. Technical report. Level 00 Revision 01.16. 2014. URL: https://trustedcomputinggroup.org/tpm-main-specification/ (visited on July 17, 2017).

[60] Stephen Tu, M Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. „Processing analytical queries over encrypted data". In: *Proceedings of the VLDB Endowment*. Volume 6. 5. VLDB Endowment. 2013, pages 289–300. DOI: 10.14778/2535573.2488336.

[61] Victor Vu, Srujay Setty, Andrew J Blumberg, and Michael Walfish. „A hybrid architecture for interactive verifiable computation". In: *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE. 2013, pages 223–237. DOI: 10.1109/SP.2013.48.

[62] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. „Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves". In: *European Symposium on Research in Computer Security*. Springer. 2016, pages 440–457.

[63] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. „Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems". In: *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*. IEEE – Institute of Electrical and Electronics Engineers, 2015. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=246400.

[64] Andrew Chi-Chih Yao. „How to generate and exchange secrets". In: *Foundations of Computer Science, 1986., 27th Annual Symposium on*. IEEE. 1986, pages 162–167.

[65] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. „CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. 2011, pages 203–216. DOI: 10.1145/2043556.2043576.

Eidesstattliche Erklärung

Hiermit versichere ich, dass meine Master's Thesis „Hardening Applications with Intel SGX“ („Anwendungen härten mit Intel SGX“) selbständig verfasst wurde und dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt wurden. Diese Aussage trifft auch für alle Implementierungen und Dokumentationen im Rahmen dieses Projektes zu.

Potsdam, den 17. Juli 2017,

_____

(Fredrik Teschke)