Hardware-Based Trusted Computing Architectures for Isolation and Attestation

Pieter Maene, Johannes Götzfried, Ruan de Clercq, Tilo Müller, Felix Freiling, and Ingrid Verbauwhede, *Fellow, IEEE*

Abstract—Attackers target many different types of computer systems in use today, exploiting software vulnerabilities to take over the device and make it act maliciously. Reports of numerous attacks have been published, against the constrained embedded devices of the Internet of Things, mobile devices like smartphones and tablets, high-performance desktop and server environments, as well as complex industrial control systems. Trusted computing architectures give users and remote parties like software vendors guarantees about the behaviour of the software they run, protecting them against software-level attackers. This paper defines the security properties offered by them, and presents detailed descriptions of twelve hardware-based attestation and isolation architectures from academia and industry. We compare all twelve designs with respect to the security properties and architectural features they offer. The presented architectures have been designed for a wide range of devices, supporting different security properties.

Index Terms—Trusted computing, security hardware, protected module architectures, isolation, attestation

1 Introduction

Computers play an important role in today's society and will become an even greater part of our daily lives in the future. However, any computing system can be breached by attackers, from complex cloud hosting infrastructures with thousands of servers to the tiny microcontrollers used in the Internet of Things (IoT). Critical software vulnerabilities have been discovered in home appliances [1], cars [2], and even industrial control systems [3]. These attacks all lead to systems that no longer behave as intended by their designers. For the exploited television sets and fridges, the consequences were rather small, as the attackers only sent out spam. However, the researchers who discovered the car vulnerability had full control over the vehicle, and Stuxnet set Iran's nuclear program back years.

The goal of *trusted computing* is to develop technologies which give users guarantees about the behaviour of the software running on their devices. More specifically, a device can be trusted *if it always behaves in the expected manner for the intended purpose* [4]. This means that even when an attacker gains control of the system, he cannot make it misbehave. This is a complex goal, covering many aspects, resulting in a wide range of solutions based on software, hardware, or co-design of both. Industry has also been active in this field, adding security mechanisms to their products, some of which can be found in millions of devices.

After defining trusted computing, the difference between it and *trustworthy* computing [5] should be pointed out. In trusted computing, users are asked to trust a set of components, and the security of the system is no longer guaranteed if any of its components are breached. Users are given no guarantees that the trusted components will not breach their security policies. On the other hand, trustworthy computing provides users with proof that its trusted components will not violate security [6]. Its focus is more on improving software engineering processes [7], rather than modifying the hardware architecture.

Although software-based trusted computing architectures with interesting results have been proposed [8], [9], [10], [11], [12], they can typically only be used in limited settings, nor are they able to give the same security guarantees as hardware-based architectures. An important part of trusted computing is to protect against attackers who have full control over the system, i.e., any application could have been exploited, as well as the Operating System (OS). Many hardware-based architectures protect applications from a malicious OS. No software-only solution can provide these guarantees, as an attacker can always manipulate software if the OS is not trusted. It is much harder for an attacker to modify hardware functionality, to the extent that hardware is considered to be immutable. Therefore, a user's trust is said to be rooted in the hardware, which is also why this paper only considers hardware-based architectures.

Trusted computing has been an active field of research over the past 10 years, and several architectures have been proposed for devices ranging from lightweight embedded systems to high-performance desktop and server processors. This large number of designs makes it difficult to determine which problems have been solved already, and how

- *P. Maene, R. de Clercq, and I. Verbauwhede are with COSIC, the Department of Electrical Engineering (ESAT), KU Leuven, Leuven 3000, Belgium, and imec, Heverlee 3001, Belgium. E-mail: {pieter.maene, ruan.declercq, ingrid.verbauwhede}@esat.kuleuven.be.*
- *J. Götzfried, T. Müller, and F. Freiling are with the Department of Computer Science, FAU Erlangen-Nürnberg, Erlangen 91054, Germany. E-mail: {johannes.goetzfried, tilo.mueller, felix.freiling}@cs.fau.de.*

Manuscript received 2 June 2016; revised 28 Oct. 2016; accepted 16 Dec. 2016. Date of publication 4 Jan. 2017; date of current version 16 Feb. 2018. Recommended for acceptance by R. Lee, P. Chaumont, R. Perez, and G. Bertoni. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TC.2017.2647955

this was done within the constraints of the target platform. To the best of our knowledge, this paper gives an overview of all major trusted computing designs making hardware changes to the underlying architecture from the academic community as well as industry, and offering either isolation or attestation. The former protects applications from other software, while the latter allows a third party to get proof that the software was not tampered with. Architectures which do not meet these requirements, being software-based designs or providing different functionality, are not included.

2 Background

This section introduces basic trusted computing terms which are most widely used by the different architectures. We assume that the reader is familiar with symmetric and asymmetric encryption algorithms, hash functions, and Message Authentication Codes (MACs). Otherwise, an introduction can be found in the Handbook of Applied Cryptography [13].

Protected Module Architectures (PMAs). Software has become incredibly complex, making it almost impossible to prove that an application does not contain bugs. Furthermore, attackers are always looking for vulnerabilities they can exploit to gain access to a system. Therefore, McCune et al. introduced the concept of Protected Module Architectures [14], where security-critical components are separated into smaller *protected modules*. Since they are much smaller, it is easier to verify their correctness. These modules are then isolated (section 4.1) from any other software on the system, so that they cannot be tampered with. It has been shown that PMAs can be implemented at any level of the architecture, from the hardware to the OS kernel [15].

Throughout this paper, we will adopt the terminology used by the original authors. Consequently, protected modules will also be referred to as *Software Modules (SMs)*, *Secure Executables (SEs)*, *enclaves*, *secure tasks*, or *trustlets*.

Trusted Computing Bases (TCBs). Are the sets of hardware and software components which are critical to their architecture's security. The careful design and implementation of these components are paramount to the overall security. The components of the TCB are designed so that, when other parts of the system are exploited, they cannot make the device misbehave. Ideally, a TCB should be as small as possible in order to guarantee its correctness.

Measuring. Is used to verify the authenticity of software components. This is done by calculating a hash or MAC of its code and data. Some designs also include other identifying information, like the memory layout. The measurement result can then be used to attest (section 4.1) the component's state. Since a hash or MAC value for a given input is probabilistically unique, it also identifies the state of the software component at that time.

Trust Chains. Are formed by verifying each component's validity from the bottom up. For software, this can be done by measuring each component in the chain before its execution.

3 Attacker Model

In general, any trusted computing architecture only protects against attackers with a specific set of capabilities. First, the attacker is assumed to be in complete control of all software on all the devices in the system, except for the software that is part of the TCB. This means that he can tamper with the OS, or even deploy malicious software components. Some architectures use software modules that are part of the TCB, and it is assumed that the attacker cannot change these.

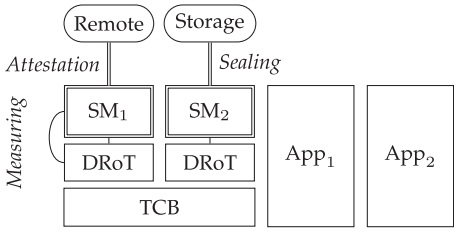

Fig. 1. In general, a Protected Module Architecture (PMA) runs multiple Software Modules (SMs) side by side, along with one or more unprotected applications. The TCB ensures that the state of the Software Modules (SMs) is protected from any other software running on the system. The measurement of the SM establishes a Dynamic Root of Trust (DRoT). The result can be used to attest the state of the module to a remote verifier. By sealing data, the SM can send it securely to untrusted storage.

Second, the attacker is assumed to be in control of the communication channel to the device. He is therefore capable of sniffing the network, modifying traffic, and mounting Man in-the-Middle (MitM) attacks. These abilities are important when considering attestation.

Third, the Dolev-Yao attacker model [16] is used, where the attacker is assumed to be unable to break cryptographic primitives, but can perform protocol-level attacks.

Fourth, none of the architectures are capable of providing availability guarantees, and therefore cannot protect against Denial-of-Service (DoS) attacks.

Fifth, architectures without memory protection consider physical attacks on the hardware out of scope. This means that the attacker does not have physical access to the hardware, cannot probe the memory, and cannot disconnect components. However, architectures which include memory protection consider the attacker capable of performing physical attacks on off-chip memory, but not on any hardware components which are part of the TCB, such as the Central Processing Unit (CPU). In addition, none of the architectures include protection against hardware side-channel attacks, which are therefore not considered in this paper.

Finally, software side-channel attacks exist where untrusted software or malicious modules target memory access patterns. Architectures which do not protect against this, therefore do not give attackers the capability to monitor cache accesses or to observe the addresses of page faults.

4 Properties

Our work discusses and compares the different architectures with respect to a set of security properties (section 4.1) and architectural features (section 4.2). The former are the result of security mechanisms which were added specifically to provide users with strong guarantees about the software executing on their system (section 1). Fig. 1 gives a schematic overview of a PMA, also illustrating some of these security properties. The latter are features commonly found in current microcontroller and general-purpose architectures, but which require special attention in the context of trusted computing.

4.1 Security Properties

The following seven security properties were selected to facilitate this paper's discussion, each of which is offered by at least one architecture described in section 5. Since this paper focuses on architectures which provide isolation and attestation, these were included first. All other properties are the result of new functionality introduced by the discussed architectures, and were added to enable a comparison between all designs. The first five are fundamental features provided by the trusted computing architectures discussed in this paper, while the last two are also more widely used in security research.

Isolation. Denotes a hardware-based architectural mechanism that provides access control for software and its associated data. By placing code and data inside a protected module, no software outside it can read or write its runtime state or modify its code. Execution of code inside such a module can only be started from a single predefined location. Such an *entry point* ensures that attackers cannot reuse the module's own code to extract secrets or implement malicious behaviour, as is done in Return-Oriented Programming (ROP) [17]. Current architectures allow for one or more modules, and some even support running them concurrently. Protected modules are used to store confidential information like secret keys, as other software cannot access its state. Writes into them are also prevented, protecting the integrity of the module's code and data.

Attestation. Is the process of proving to an authorized party that a specific entity is in a certain state. In order to give strong security guarantees, an architecture which supports attestation should guarantee integrity of the attested state as well. Trusted computing architectures may provide *local* and *remote* attestation. The former refers to one software module attesting its state to another running on the same platform, while the latter attests to a remote party residing outside the trusted system.

A common way to implement attestation is to measure (section 2) software modules during their initialization, while preventing later modifications by means of isolation. It can then be used to authenticate a challenge sent by the authorized party, as the measurement uniquely identifies the state of the module. Since it could only have resulted from measuring a specific software module in a certain configuration, the authorized party knows it communicates with this module.

Sealing. Wraps confidential code or data in such a way that it can only be unwrapped under certain circumstances. Code or data are wrapped by binding it either to a specific device, a certain configuration of the device, the state of a software module, or a combination of these. It can then only be unwrapped when the binding conditions are met, e.g., on the same device or one which runs the same configuration. Sealing is usually based on encryption, and relies on similar mechanisms as software attestation, i.e., the key for encrypting confidential code or data is typically derived from the software module measurement taken during initialization.

Dynamic Roots of Trust (DRoT). In order to keep the TCB (section 2) as small as possible, most trusted computing technologies build trust chains. However, these chains always need to be anchored in a component that is inherently trusted, which are referred to as Roots of Trust (RoT). A Dynamic RoT (DRoT) is established for a software module at runtime, measuring (section 2) the application's state right before execution starts. It is typically combined with isolation to protect against Time-of-Check Time-of-Use (TOCTOU) vulnerabilities as well, where an attacker changes the module's code after it has been measured.

Code Confidentiality. A trusted computing architecture which guarantees code confidentiality ensures that sensitive code or static data cannot be obtained by untrusted hardware, software, or any other unauthorized entity covered by the attacker model. This property usually requires both isolation and encryption. Isolation is used to protect against software attackers after modules have been loaded. Encryption is needed to protect against attacks targeting confidential information before loading, and to prevent physical attacks. Sealing can be used to ensure that only a certain software component can obtain some Intellectual Property (IP).

Side-Channel Resistance. A trusted computing architecture is called side-channel resistant with respect to software attackers if no software module, including privileged software like an OS, is able to deduce information about the behaviour of other modules apart from their Input/Output (I/O). Specifically, information flow through untrusted channels, such as caches, or information revealed by page faults cannot leak to untrusted software. An architecture with side-channel resistance should take care to flush caches during context switches. Information leakage due to page faults, for example, can be prevented by giving each software module the ability to handle its own page faults.

Memory Protection. Specifically refers to protecting the integrity and authenticity of data sent over system buses or stored in external memory from physical attacks. We consider both *passive* (e.g., bus snooping) and *active* (e.g., fault injection) attacks. First, this means that data has to be encrypted, to prevent sensitive data from leaking. Second, it also has to be integrity-protected, for example, using a MAC. Third, replay attacks, where previously valid memory contents are restored, have to be prevented as well. These operations have to be performed when data is sent to or fetched from external memory.

4.2 Architectural Features

Designers have to make numerous decisions when integrating the security mechanisms needed for trusted computing in complex modern processor architectures. We selected seven basic features they typically take into account. The first, targeting a lightweight architecture or not, is special as it will also influence the design of the other features.

Lightweight. We define an architecture to be lightweight when it does not use a Memory Management Unit (MMU). Lightweight embedded systems have very simple memory hierarchies, and therefore do not need complex memory management. Furthermore, they only run a limited amount of applications, which share the memory space cooperatively, not requiring virtual addressing to map the memory.

Coprocessor. Many trusted computing architectures require security mechanisms to be implemented in hardware. In case of a coprocessor, this functionality is added as a separate chip or module inside the same System-on-Chip (SoC) that interfaces with the main processor. Other approaches are to integrate the functionality inside the processor, or by adding the functionality as a component to a SoC. Trusted computing architectures that are integrated in

a processor can typically provide stronger security guarantees than coprocessor-based designs, because data does not have to leave the CPU. Some functionality also cannot be implemented in a coprocessor, e.g., isolation.

Hardware-Only TCB. It is typically better for the TCB (section 2) to rely on hardware, as this provides stronger security guarantees, such as protection from an untrusted OS. In this work, we only discuss architectures that have a hardware-only TCB or architectures that have a TCB consisting of hardware and software components.

Preemption. When preemption is supported, the system can suspend running tasks at any time, without first obtaining permission from the task. This makes it possible to handle interrupts, but also allows preemptive scheduling of multiple protected modules. Preemption mainly impacts the context switching logic, since the architecture now has to ensure that no sensitive information can leak between modules, as this would violate the isolation primitive. Without support for preemption, applications have to run cooperatively, i.e., they need to call each other after finishing a task.

Dynamic Layout. A static layout is often used when all software shares the same address space, and no MMU is present to provide virtual memory for different applications. It has the disadvantage that one trusted entity, e.g., the hardware or software manufacturer or a system integrator, needs to compile all software and fix the layout before deployment to the target device. With a dynamic layout, however, applications can be loaded to locations that do not need to be known at compile time.

Upgradeable TCB. Architectures which have a HW-only TCB are not upgradeable, because its components can no longer be changed after being manufactured. However, some designs include trusted software components, typically to implement functionality which would be too expensive in hardware. These components are then protected by a hardware mechanism, such as Programmable ROM (PROM). This not only results in more design flexibility, but also enables upgrading the TCB at a later time, e.g., when a bug has been discovered or to add new functionality.

Backwards Compatibility. When adding features, an important consideration for industry is whether legacy code runs on the modified architecture without any changes, possibly after recompilation. Since these applications do not use the introduced security mechanisms, they typically do not receive any of the associated security guarantees.

5 Architectures

This section gives detailed descriptions of twelve isolation and attestation designs, which make hardware modifications to their target platform. Architectures which are implemented entirely in software or provide other functionality were therefore not included. A wide range is covered, from lightweight designs for the IoT to desktop and server architectures. The selection was not limited to academic research, but also includes industry efforts. All architectures were ordered chronologically by year, and then alphabetically.

5.1 AEGIS

Suh et al. designed AEGIS [18] in 2003 already, making it one of the oldest trusted computing architectures. It provides programs with a Tamper-Evident Environment (TE), where any memory tampering, either from software or physical, is detected. Even stronger guarantees are provided by Private and Authenticated Tamper-Resistant Environments (PTRs), which also protect the confidentiality of code and data. The CPU itself is considered to be trusted, placing external memory and peripherals outside of the TCB, where they are vulnerable to software and hardware attacks. In a hardware-supported implementation, the OS can be malicious, but when a *Security Kernel (SK)* is used to implement AEGIS' functionality, parts of the OS need to be trusted.

Since the system can also run legacy code, the protection mechanisms have to be enabled by calling the `enter_aegis` instruction. This will calculate a hash of the program and store it in a protected area. The program hash will also be used later as an identifier. Each program also includes some code which will measure any other data or code it depends on. Finally, it should also ensure that it is running in the expected environment (e.g., the current CPU mode). After `enter_aegis` has been called, the program is isolated, and any memory tampering will be detected. Since the on-chip caches are considered trusted, the hardware only needs to prevent applications from writing to locations they do not have access to. This is done by tagging the cache entries with a Secure Process ID (SPID), which is assigned when the program is started.

However, a more complex mechanism is needed to protect off-chip memory. Whenever data is read into the cache, its integrity needs to be verified. When a cache block is evicted, the corresponding leaf node of a hash tree is updated with the new contents. Of course, this means that all internal nodes also have to be updated. Depending on the memory size, this could result in a large number of additional transfers, and the internal nodes are therefore also cached in the L2 cache, performing updates first in the cache, and not directly in memory. The authors distinguish between non-blocking and blocking instructions. For the former, the integrity verification can be delayed, as long as the CPU is eventually notified it was working with tampered data. When the AEGIS mode is set to PTR, the CPU also needs to guarantee confidentiality. This is done by encrypting the blocks with AES in the CBC mode of operation, using 32 bit random Initialization Vectors (IVs) to guarantee uniqueness of the ciphertexts. The architecture uses separate keys for static and dynamic data. The former is used to decrypt the binary's data and code, while the latter protects any data generated at runtime.

The remote attestation mechanism hashes the provided data together with the program hash, and asymmetrically signs the result with a private key specific to the CPU. This binds the data to the code of the program, as well as the specific processor it is executing on. In case AEGIS' features were implemented in a software SK, the hash of the kernel is also included in the attestation result. This functionality is provided through the `sign_msg` instruction.

5.2 Trusted Platform Module (TPM)

The Trusted Platform Module version 1.2 [19] was specified by the Trusted Computing Group (TCG) in 2011. It is a coprocessor on the motherboard, which is capable of storing keys and performing attestation. It is a passive piece of hardware, meaning that software can interact with the

TPM, but needs to do so explicitly. To give a local or remote party guarantees, any software that runs on the target device, i.e., the boot loader, OS, and applications, needs to be measured successively by the TPM, and is consequently part of the TCB. Code manipulation is only detected during measurement, and all parts of the software are considered trusted after loading. The TPM only provides limited protection against physical attacks, because not only the CPU package, but the TPM chip and all connecting buses are part of the TCB as well. For instance, a physical attacker can compromise software integrity by tapping the Low Pin Count (LPC) bus between the TPM and CPU [20].

The design principles published by the TCG in version 1.2 specify the minimum functionality and cryptographic primitives that are required for TPMs, but a TPM manufacturer is allowed to extend the hardware module with additional functionality. Each TPM has to be equipped with a Random Number Generator (RNG) as a randomness source and an RSA implementation with at least 2048 bit keys. The minimum requirement for software measurement is the SHA-1 hash algorithm. At manufacturing time, the Endorsement Key (EK) is generated and written to persistent memory within the TPM. The EK is unique to every TPM chip, not known to the user, and serves as master key for all operations provided by the TPM. In addition to the EK, Attestation Identity Keys (AIKs) and storage keys can be generated on the fly and stored in volatile memory within the TPM. AIKs are used for all operations directly involving digital signatures, whereas the storage keys are used for encryption and decryption of data. Finally, a TPM contains a certain number of Platform Configuration Registers (PCRs), which are capable of storing successive hash values for code or data that is sent to the TPM and are important for remote attestation. The TCG also specified TPM version 2.0 [21] in 2014, supporting a larger variety of cryptographic algorithms, multiple banks of PCRs, and three hierarchies instead of only one. For simplicity, we focus on version 1.2 in this paper.

With the minimum functionality required by the TCG, each TPM is able to perform at least three different operations. First, it is able to bind code or data to a given device by encrypting it with one of the storage keys. The encrypted code or data can then only be decrypted on the same platform, because the particular storage key cannot be extracted from volatile memory within the TPM.

Second, a TPM is able to attest to another party that the given device is currently running a certain software configuration (section 4.1). To this end, code or data sent to the TPM is hashed together with values of specific PCRs, and the result is again stored in the same PCRs. This way, each software component running on the device is able to successively extend a measurement over all components running on the device. The PCRs are initially set to a fixed value before starting the system, and because the hash function is irreversible, it is not possible to set the PCRs to a user-defined value. The resulting hash values from the PCRs can then be cryptographically signed by an AIK, and this signature can later be verified by a second party. When this party receives a correctly signed value, it can be sure that the system runs a certain software configuration, because this signed message could not have been created without going through the software measuring process.

Third, code or data can be sealed to a given device in a certain software configuration (section 4.1). In general, sealing works similar to binding, i.e., code or data is encrypted and decrypted, but it is additionally ensured that sealed code or data is only decrypted if the platform configuration has not changed in between. To check against changes of the platform configuration, the PCR values are saved together with the sealed code or data and checked against the current PCR values during unsealing.

Attestation and sealing only behave as intended if the platform configuration is measured from the earliest boot step, up to the currently running software component, because otherwise malicious software could potentially exclude itself from the measurement. This restriction is the biggest disadvantage of the standalone TPM over other solutions that support DRoTs (section 4.1).

To overcome the restriction of all software having to be part of the TCB, Intel introduced its Trusted Execution Technology (TXT) [22], which also uses the TPM chip, but allows dynamically establishing a new Root of Trust for software running in a virtualized environment besides the usual software stack. TXT ensures that the virtualized software has exclusive control over the device by suspending all other running software, i.e., the OS and all applications. When switching to trusted TXT software, the CPU essentially performs a warm reset and initializes a certain subset of PCRs with a new value. The TXT software can then extend this measurement and attest to a second party that it has not been modified before being loaded. Since it monopolizes all resources once it has been loaded, the integrity of the TXT software is guaranteed over its entire runtime.

Although TXT can be used to overcome the restriction of all software having to be part of the TCB, it still has some issues. Suspending all other applications on the device for the TXT software to run negatively impacts performance, or might even lead to losing interrupts depending on its size. Since the TXT software has to run exclusively, it cannot easily use functionality of untrusted software and needs to perform expensive context switches. Finally, all physical attacks that succeed for a standalone TPM, e.g., LPC bus tapping, also succeed for TXT. Fides [11], Flicker [14], and TrustVisor [23] are examples of architectures which build on the functionality offered by TXT.

5.3 TrustZone

GlobalPlatform wrote an industry standard for security architectures called the *Trusted Execution Environment (TEE)* [24], [25]. The TEE is a secure area of the main processor, and provides isolated execution, integrity of trusted applications, as well as confidentiality of trusted application resources. The TEE is isolated from the Rich Execution Environment (REE) where the untrusted OS runs. The REE resources are accessible from the TEE, while the TEE resources are not from the REE, unless explicitly allowed. Therefore, only trusted resources can access other trusted resources. The standard does not specify how manufacturers should implement it. TrustZone is an implementation of this standard by ARM.

TrustZone [26] is a hardware-based security architecture for a SoC that is currently used in a large number of smartphones. The TEE, also called the *secure world*, provides

protection for trusted hardware and software resources. Hardware-based mechanisms ensure that resources in the REE's untrusted OS, or *normal world*, cannot access secure world resources. This is done by two main hardware features. First, the SoC's AXI bus ensures that secure world resources cannot be accessed from normal world resources. Second, a TrustZone-enabled processor core uses time-slicing to execute code in either the secure or normal world.

To enforce isolation between trusted hardware resources on the bus, a control signal known as the Non-Secure (NS) bit was added to the AXI specification. This bit is used to communicate the security state of a master component to a slave component. The bus or slave logic uses this bit to ensure that the security separation is not violated. When an untrusted master attempts to access a secure slave, the transaction should fail and an error may be raised.

A TrustZone core can switch between security states at runtime. When the processor core is in the secure state, it generates AXI transactions with the NS bit set to zero, allowing it to access resources in both security domains. However, a processor core in the normal world can only access normal world resources. The processor's L1 and L2 caches use a bit to store the security state of the transaction that accessed the memory. The cache controllers are then assumed to be responsible for ensuring that only secure masters can access memory that was fetched from a secure slave. Extending the cache removes the need for a flush when performing a context switch between security domains, and further allows software to efficiently communicate from the non-secure to the secure world.

To perform a context switch to the other world, the processor first has to pass through a new mode called *monitor mode*, which serves as a gatekeeper that manages context switches between the two worlds. This is done by saving the state of the current world and restoring the state of the world being switched to. Monitor mode exists in the secure world, and both privileged and user mode exist in each world. Normal world entry to monitor mode is only possible via an interrupt, external abort, or explicit call of the `smc` instruction. Secure world entry to monitor mode can additionally be invoked by writing into the Current Program Status Register (CPSR). ARM recommends to execute monitor code with interrupts disabled. The address mappings in the MMU can be configured independently for each world. This allows the OS in each security domain to enforce its own memory management. Inter-Process Communication (IPC) with small messages can be done by placing the message inside registers and then invoking `smc`. For larger messages, it is possible to use shared memory.

Interrupts can be serviced in either security domain. When a context switch is required to handle an interrupt, the processor traps directly into monitor mode. A different exception vector table is used to specify the interrupt service routine addresses for normal world, secure world, and monitor mode respectively. Each of the vector table base addresses can only be updated from its respective mode. This enables secure interrupt sources that cannot be manipulated by normal world software.

During the boot process, a chain of trust is formed by verifying the integrity of the trusted second stage boot loader and trusted OS before execution. The TrustZone processor starts in secure world when it is powered on. The firmware of the first stage boot loader is implicitly trusted, and is typically located in ROM. It initializes critical peripherals, such as memory controllers, and further performs an integrity check of the second stage boot loader, which is stored in flash. If this check passes, the second stage boot loader is executed. It in turn verifies the integrity of the secure world OS and boots it, after which the normal world OS is started without performing an integrity check. Some implementations of the secure OS also verify the integrity of trusted applications before loading them. TrustZone uses a signature scheme based on RSA. A vendor uses its private key to sign the code. The firmware then uses the public key to verify the signature at runtime. In order to support multiple different vendors, the architecture supports the use of several public keys.

Since any trusted component can violate the system's security, it is important to respect the principle of least privilege and restrict the access of each component in the TCB. TrustZone violates this design principle, as applications from different vendors run in the same secure world. Furthermore, when multiple secure master devices from different vendors are placed on a TrustZone SoC, least privilege is violated as all the secure masters have access to all memory.

5.4 Bastion

Bastion [27] is a combined hardware-software architecture, which relies on a trusted hypervisor together with a modified processor to ensure confidentiality and integrity for security-critical software modules. Physical attacks on all hardware components apart from the CPU package are allowed, i.e., Bastion provides memory protection. Only single-core processors are currently supported.

Since everything apart from the microprocessor and the hypervisor is considered untrusted, including firmware and code needed during booting, Bastion first protects the state of the hypervisor. Afterwards, the hypervisor is able to protect any number of software modules. To this end, the Bastion hypervisor calls `secure_launch` from its initialization routines, which computes a cryptographic hash over the state, i.e., code and data, of the hypervisor and stores the result in a CPU register. The `secure_launch` routine also generates a new key which is used to encrypt and integrity-protect all data belonging to the hypervisor with the help of an on-chip crypto engine. The hash value later serves as the identity of the hypervisor and is, for example, needed to unseal permanently stored data. The implementation of `secure_launch`, as well as the register contents, cannot be modified from software.

After the hypervisor has been loaded, software modules can invoke a new `secure_launch` hypercall for initialization, which activates runtime memory protection for all module pages and computes a hash of the module's initial state, including the virtual memory layout, that serves as the module's identity. For instance, this identity is used to seal data that needs to be permanently stored on disk.

A modified CPU ensures that the hypervisor is invoked for each Translation Lookaside Buffer (TLB) miss. The hypervisor checks whether the virtual address responsible for the access corresponds to the one associated with the physical page and a specific software module. For these

checks, modules are identified by a unique identifier (usually eight to 20 bits), which is assigned during the `secure_launch` call. All untrusted software not belonging to a specific module is treated as a module with identifier zero, ensuring that untrusted software cannot access code or data from security-critical modules.

To invoke a function of a secure module, a special `call_module` hypercall is added, which takes the hash of the destination module and the entry point as parameters, because direct transitions would trigger a memory violation. Similarly, on returning, the `return_module` hypercall is needed. When a module needs to be preempted, for example, due to a timer interrupt, the hypervisor takes care of saving all state information, such as register contents, and wiping sensitive data before calling the interrupt handler. When returning from the interrupt handler, the hypervisor also takes care of restoring all state information and handing back control to the secure module.

5.5 SMART

El Defrawy et al. designed SMART [28] to establish a DRoT in remote embedded devices (section 4.1). The architecture is also minimal, requiring only the smallest possible set of hardware changes in order to implement remote attestation, which was later formalised by Francillon et al. [29]. It was one of the first designs using hardware-software co-design to build a lightweight trust architecture. Prototypes to demonstrate the feasibility were built on open-source clones of the ATmega103 and openMSP430. The attacker model specifically assumes that adversaries do not tamper with ROM. Any peripherals which can directly access memory should also be disabled while SMART is executing.

In general, SMART provides remote attestation of a memory range $[a, b]$ specified by the verifier. It then calculates a MAC over this memory region and sends the result back. The verifier calculates the same MAC over the expected contents and compares both. This process dynamically established a RoT. In addition, an address x can be given, where execution will continue atomically after SMART has completed. By choosing $x = a$, the verified code is started.

Support for SMART requires four features: attestation Read-Only Memory (ROM), secure key storage, MCU access controls, and reset and memory erasure. The ROM stores the attestation code which is invoked when a verification is requested. This program disables interrupts, measures the specified memory region by calculating a SHA-256 HMAC, and reports the result. When x is set, interrupts remain disabled and control jumps to that address, but otherwise they are re-enabled and execution continues.

Secure key storage is added to the microcontroller for the symmetric key used to calculate the HMAC, and the MCU access controls ensure that it is only accessible when the CPU's Program Counter (PC) is in the ROM region containing the attestation code. In order to prevent code reuse attacks, the MCU also enforces a single entry point into the ROM code, only allowing access from the initial instruction, and disabling exits from any instruction other than the last. When an invalid memory access is detected by either mechanism, the processor is reset immediately.

The attestation code is carefully written to ensure it cleans up any sensitive data after it has finished. However, when the processor is reset during its execution, this cleanup might be skipped. Therefore, all memory is erased by the hardware when the processor boots or after a reset.

SMART and TrustLite (section 5.10) were later used to prototype a scalable attestation mechanism for large swarms of small embedded devices [30].

5.6 Sancus

Sancus [31] is a hardware-only PMA designed by Noorman et al. for lightweight embedded devices, like wireless sensor nodes. In addition to isolating an application's code and sensitive data, it also has support for remote attestation. It adds secure linking functionality as well, enabling applications to verify modules they depend on.

The architecture was designed for small embedded devices, which are typically deployed in large swarms. These *nodes* are managed by an Infrastructure Provider (IP), and they share a fixed key K_N with it, which is etched into the silicon. When Software Providers (SPs) want to load a protected application onto a node, they have to go through the IP. Each SP is assigned a unique public identifier, which is used to derive a key $K_{N,SP} = kdf(K_N, SP)$ for the SP. The kdf is implemented in hardware to realize a zero-software TCB. Since the IP manages the node key K_N, it knows all other keys used in the system, and SPs therefore have to trust it to behave as intended.

In Sancus' system model, protected applications are called *Software Modules (SMs)*. Each SM consists of a *text section*, which contains code and constants, and a *protected data section* where sensitive dynamic data can be stored. Additionally, an SM can include *unprotected sections*, which makes it possible for developers to keep the size of the sensitive code as small as possible. Each SM is also assigned an *identity*, consisting of the contents of its code section, and its layout. The latter are the start and end addresses of its protected code and data sections, making it possible for two modules where these sections are identical to exist on the system at the same time. Similar to $K_{N,SP}$, the SM's identity is used to derive a third-level key $K_{N,SP,SM} = kdf(K_{N,SP}, SM)$.

When an SM uses functionality from another protected module, it can use *caller authentication* to ensure that the other module was not tampered with. To this end, SM_1 stores a MAC with its own key K_{N,SP,SM_1} of SM_2's identity in an unprotected section. It can issue a special instruction at runtime to verify the MAC. The hardware also enforces a single entry point. This single physical entry point is not a limitation, as it can be multiplexed to multiple logical ones.

The memory access control mechanism is program counter-based, i.e., the access rights depend on the current value of the processor's program counter. The protected text section is always readable, but only executable when the module is executing. The only exception is the entry point, which is always executable. The text section has to be readable for the caller authentication to work, as it is included in the MAC. Similarly, the protected data section is only read- and writeable when the program counter is in the module's text section.

These protection mechanisms are enabled by calling the special `protect` instruction, which has the layout information and SP identity as parameters. This instruction also derives the node key $K_{N,SP,SM}$ and stores it together with

the layout in a *protected storage area*, inaccessible from software. Invoking `unprotect` disables the memory protection. Memory violations are handled by resetting the CPU, at the cost of availability, which is excluded from the attacker model (3). Three additional new instructions are introduced for the remote attestation and secure linking functionality respectively. First, `MAC-seal` can be used to calculate a MAC with $K_{N,SP,SM}$ over a given memory range. Second, calling `MAC-verify` will calculate the MAC of the specified module, with the current module's key, and verify that it matches the MAC stored in memory. Third, since the MAC calculation is expensive, a module is assigned a monotonically increasing ID by the CPU at load-time, which can be queried with `get-id`. This ID is used to securely link to the same module at a later time, by storing it and checking that it still matches the one returned by `get-id`.

A prototype was developed based on the OpenMSP430, an open-source implementation of Texas Instruments' MSP430 processor. The most important changes are the addition of the on-chip protected storage area, the Memory Access Logic (MAL) circuitry, and a hardware implementation of HMAC based on SPONGENT [32], a lightweight hash function. The number of SMs which can be loaded concurrently is chosen at synthesis time, and determines the size of the on-chip memory. The MAL circuit is fully combinational, so it does not need additional cycles to perform its checks. Furthermore, it is shown not to be on the processor's critical path, meaning that it doesn't impact the clock frequency. In addition to the hardware changes, an LLVM-based toolchain to compile SMs was also built. This allows developers to easily use the new functionality by annotating their code, and inserts stubs which handle stack switching, secure linking, and entry point multiplexing. Evaluation of the prototype showed that the main performance overhead is found in instructions which use the hashing functionality, i.e., `protect`, `MAC-seal`, and `MAC-verify`. The duration of the hash calculation depends on the size of the input data.

Soteria [33] is an extension of Sancus which takes advantage of the architecture's functionality to add code confidentiality. This is done by creating a loader module SM_L, which has the module key K_{N,SP_L,SM_L}. This loader module atomically reads the encrypted binary from the node's memory, and writes the decrypted code back to memory, after which it calls `protect` on the decrypted SM. Each module is assigned an identity $\widetilde{SM_E}$, which is used to derive the encryption key $E_{SM_E} = kdf(K_{N,SP_L,SM_L}, \widetilde{SM_E})$. Although the encryption algorithm is implemented in software, it is not part of the TCB, because the key derivation includes the module's text section. Therefore, any changes to SM_L would result in a different E_{SM_E}, which would be detected during authenticated decryption. Due to the way the derivation of E_{SM_E} is implemented, SM_L and SM_E mutually authenticate each other. Additionally, Sancus' hardware is modified to deny other SMs access to the text section of SM_E.

5.7 SecureBlue++

SecureBlue++ [34], [35] is an early PMA from IBM, which isolates Secure Executables from each other, and protects the confidentiality and integrity of their data and code. The main architectural changes involve a Memory Protection Unit (MPU), using different mechanisms at each level of the memory hierarchy. It also protects against physical memory attacks, as well as the introduction of new instructions.

An SE binary consists of a cleartext section containing a loader, which copies the cleartext integrity tags of the compiled binary into memory, and then calls the new `esm` instruction to start decryption of the encrypted sections and jump to the SE's entry point. The loader is followed by system call pass-through buffers and the *Executable Key*, which was used to encrypt the sections that are stored after it, namely the metadata for the call to `esm`, and the application's data and code. The Executable Key itself is encrypted asymmetrically under a public key bound to the CPU that the application will run on. The private key is installed in the factory, and the manufacturer signs the public part to generate a certificate asserting its validity.

The MPU protects an application's data and code at all levels of the memory hierarchy. Encryption is used to protect data in external memory, automatically decrypting cache lines and verifying their integrity when they are read from memory. Similarly, evicted lines are encrypted on the fly, also updating the integrity protection tree. A tree is used because replay protection requires a nonce, which in turn needs to be stored and MACed. All integrity information is stored in a dedicated memory region, with only the root node and its metadata requiring expensive on-chip storage.

The caches store everything in plaintext, and therefore also need to enforce access control. This is done by adding a label to each cache line with the ID of the SE it belongs to. The CPU stores the current Secure Executable ID (SEID) in a special register and compares it to the line's label. Instead of storing the SEID directly, a Memory Region ID (MRID) is used. This is an index into the Protected Memory Region (PMR) table, which holds the metadata of a specific page, like the owner's SEID. This table also manages shared memory regions, adding a second SEID for the first sharer. Any additional SEs requiring access are given the same secret, which needs to be present at a specific memory location.

To avoid leaking the values stored in registers after a context switch, SecureBlue++ stores the registers protected in the cache, which means they would also be encrypted automatically when evicted to memory. A new privileged instruction, `RestoreContext` can be used by the OS to restore the registers and wake the previously active SE, which is indicated in a special register.

Since the isolation mechanism prevents even the OS from accessing an SE's data, two approaches for handling syscalls are presented. The first is modifying *libc* to wrap the system calls, not requiring any hardware support. The second is to change the behaviour of the syscall instruction `sc`. Before transferring control to the OS, the CPU checks whether it is in SE mode. If so, the call is redirected to a small wrapper inside the Secure Executable (SE) before invoking the `sesc` instruction, which bypasses the security check and immediately calls the OS's syscall handler.

5.8 Software Guard Extensions (SGX)

In 2013, Intel announced SGX [36], which enables establishing dynamic RoTs inside regular applications, without monopolizing the system (section 5.2). SGX supports protecting an application's code as well as data [37], is able to

guarantee integrity, and provides local and remote attestation [38]. In addition, SGX includes physical attacks on communication channels and main memory in the attacker model.

In SGX, the protected parts of an application are placed within so-called *enclaves*. An enclave can be seen as a protected module within the address space of a given process, and enclave accesses obey the same address translation rules as those to usual process memory, i.e., the OS and the hypervisor are still in charge of managing the page tables. This has the advantage that SGX is fully compatible with existing memory layouts, usually configured and managed by an MMU, and also works well in a multi-core environment. Although an enclave resides in the usual process address space, there are certain restrictions in enclave mode. For example, system calls and instructions that would cause a trap into the OS or hypervisor, such as `cpuid`, are not allowed and it is necessary to leave enclave mode before dispatching them. Furthermore, this mode can only be entered from user mode, which essentially means enclaves can only be used within applications, but not the OS [39].

An SGX-enabled CPU ensures in hardware that non-enclave code, including the OS and potentially the hypervisor, cannot access enclave pages. Specifically, a region called the Processor Reserved Memory (PRM), which contains the Enclave Page Cache (EPC) and the Enclave Page Cache Map (EPCM), is protected by the CPU against all non-enclave accesses. The EPC stores enclave pages, i.e., enclave code and data, while the EPCM stores state information about the pages currently held within the EPC. The state information consists of the enclave page access rights and the page's virtual address when the enclave was created, amongst others. For each (non-cached) access of an EPC page, the current access rights and virtual address are checked against the state stored within the EPCM, and if a mismatch is detected, access is denied. The caching of state information is necessary, because all software, including at system level, is considered untrusted, and therefore attacks such as enclave layout changes through remapping have to be prevented directly in hardware. If the capacity of the EPC is exceeded, enclave pages might be written out to a memory region outside the PRM by the OS, but are then transparently encrypted with the help of a hardware Memory Encryption Engine (MEE), which is inside the CPU package.

Before an enclave can be used, it has to be created and initialized by untrusted software. The hardware ensures that an enclave's pages can only be modified before initialization is finished. All page contents, including code and static data, are measured during initialization. As this measurement depends on all contents of the enclave, and later modifications are prevented, it can be used as a basis for local or remote attestation. All operations involved in the management of an enclave, e.g., enclave creation, initialization, and destruction, are performed by system software (ring zero), while entering and leaving the enclave is done by the application software (ring three). The latter is implemented similarly to system calls, i.e., an enclave has its own execution context, there is a single entry point into the enclave, and dedicated instructions need to be called. Upon leaving the enclave, the context of the current thread is saved within an EPC page and all registers are cleared. The appropriate context is loaded again when the enclave is entered. If an interrupt occurs during enclave execution, an Asynchronous Enclave Exit (AEX) is performed by the CPU, which also saves the current enclave execution context and ensures that no data leaks to the untrusted system software handling the interrupt.

Within an enclave, other features are provided in addition to confidentiality and integrity of code and data. One enclave can attest to another that it has been loaded as intended by sending a *report*. The report includes information about the enclave (the measurement) or enclave author. This process is called *local attestation*. With the help of a trusted *quoting enclave* provided by Intel, the report can be wrapped into a *quote*, converting the local attestation to a remote attestation by signing the quote with an asymmetric attestation key, which is part of Intel's Enhanced Privacy Identifier (EPID) group signature scheme [40]. The quote can be verified by a remote party with the corresponding verification key provided by Intel. Besides local and remote attestation, data produced within an enclave can also be *sealed* to the enclave and, for example, written to memory outside the PRM. Sealed data can serve as permanent storage and retains information during different runs of an enclave. Local attestation, remote attestation, and sealing all rely on the non-forgeability of the initial enclave measurement.

Intel uses dedicated enclaves for complex functionality which would be expensive to implement in hardware, like the asymmetric cryptography needed for remote attestation. It also provides a *launch enclave* required to launch any other enclave, a *provisioning enclave* to initially provision asymmetric keys for attestation to end-user devices, and the previously mentioned *quoting enclave* to cryptographically sign the attestation quotes. The downside of this approach is that Intel has a de facto monopoly regarding enclave signing and remote attestation, as Intel decides which enclaves are allowed to run and everybody who wants to verify quotes needs to have an agreement with Intel.

More details about SGX in general can be found in a recently published exhaustive summary [41]. So far, we know of two academic solutions which use SGX in an untrusted cloud context, namely Haven [42] and VC3 [43]. However, neither solution used real hardware, but relied on an emulator instead. Finally, AMD recently presented security extensions for their processors called Secure Memory Encryption (SME) and Secure Encrypted Virtualization (SEV) [44]. The former adds memory encryption at page granularity to protect data in memory. The latter relies on this unit to isolate virtual machines from each other as well as the hypervisor.

5.9 Iso-X

Iso-X [45] is an isolated execution architecture where memory can be assigned dynamically to *Untrusted* and *Trusted Partitions*, which contain *compartments*. These compartments are essentially protected modules, and a developer can indicate which parts of his code should be compartmentalized. The architecture also includes a remote attestation mechanism in hardware, which is based on asymmetric signatures.

A static memory region is allocated and protected during boot to store hardware-maintained management information. The Physical Page Compartment Membership Vector

(CMV) is a bit vector tracking whether a memory page was already assigned to a compartment or not, while the Compartment Table (CT) records the compartment's characteristics, like its base address and size. In addition to these static structures, each compartment is also assigned a Compartment Page Table (CPT), which maps virtual to physical addresses, since the OS's page tables cannot be trusted. Finally, each compartment is also assigned a Compartment Metadata Page (CMP), which keeps track of any other data, like the compartment's public key.

Six new instructions are added to the processor, which are called either by the OS or applications to manage and use compartments. First, a new system call can be invoked by an application when it wants to start a compartment. It looks for an unused compartment identifier, and then executes the `COMP_INIT` instruction to signal the hardware to prepare its internal data structures. Similar to SGX (section 5.8), memory pages are only added to the compartment after its initialization. This is done through the `CPAGE_MAP` instruction, which also adds the page's hash to the compartment's measurement. Note that this also includes the virtual page number and permission bits. Analogously, `CPAGE_REVOKE` removes a page from the compartment, which is considered destroyed once no more pages belong to it. Finally, a compartment can be entered at its entry point through `COMP_ENTER`.

`COMP_ATTEST` generates a certificate that can be used to prove the compartment's integrity to an external verifier. This certificate is signed with the CPU's private key, which is generated in the factory. Finally, `COMP_RESUME` restores the compartment's state after a context switch, copying CPU registers back and returning to compartment mode.

The authors also present a secure swapping mechanism to support memory management. Before giving the OS's swapping mechanism access to a page, `COMP_SWAP_PREP` hashes it and overwrites the corresponding CPT entry with the result, also resetting the valid bit. The page is then encrypted symmetrically, and the corresponding CMV bit is cleared. Afterwards, the OS uses `COMP_SWAP_RET` to copy the page back to memory, decrypting it and verifying the hash. This instruction also re-enables memory protection.

5.10 TrustLite

TrustLite [46] is a generic PMA for low-cost embedded systems which was developed by the Intel Collaborative Research Institute for Secure Computing. A *trustlet* isolates software components, providing confidentiality and integrity guarantees for both its code and data. The architecture provides OS-independent isolation of trustlets, attestation of trustlets, trusted inter-process communication, secure peripherals, and supports interrupts. It was implemented as an extension to the Intel Siskiyou Peak research platform. The standard attacker model is used, with the assumption that the trustlets and bootstrapping routine behave correctly.

When the TrustLite device is booted, the first software to execute is the *Secure Loader*. It is stored in PROM, which is part of the SoC. The Secure Loader is responsible for loading all desired trustlets and their data regions into on-chip memory. In addition, it configures an MPU to enforce isolation of each trustlet's memory regions, which can include Memory-Mapped IO (MMIO) peripherals. The configured regions are also recorded in a *Trustlet Table* for use by individual trustlets or attestation routines. After the Secure Loader has configured all trustlets, the untrusted software, such as the OS, is allowed to execute. The Secure Loader is only active during initialization, and the MPU is also used to protect it at that time. As the initialization code configures the memory protection regions on platform reset, there is no need to clear the main memory as in SMART (section 5.5).

The MPU uses registers to store multiple different protection regions for each trustlet. The architecture uses program counter-based isolation. The memory regions of a trustlet are only accessible by a specific trustlet when the PC is in its code region. When the PC is outside this region, the memory regions specified in the MPU are not accessible. The processor raises an exception on an access violation. In addition, it invalidates currently executing instructions, and flushes the processor's pipeline stages.

To support interrupting an executing trustlet, the architecture needs to ensure that no information leaks. It does this by storing the current state of the processor on the stack of the interrupted trustlet, saving the stack pointer in the trustlet table and clearing the general purpose registers, after which the OS stack pointer is restored, followed by execution of the Interrupt Service Routine (ISR). A return from an interrupt is performed by jumping to the entry point, and restoring the trustlet's stack pointer in software.

Each trustlet uses an entry vector to specify the addresses which can be called by other tasks or trustlets. The trustlet itself can execute its entire code section, but other tasks or trustlets can only execute the addresses listed in the entry vector. The entry vector should be carefully programmed to avoid information leakage or other exploits.

Signalling and sending short messages are done by calling the entry of a trustlet and passing the arguments in CPU registers. Large messages can be communicated by signalling with a pointer to a shared memory region, which needs to be inside an MPU region. Trusted communication between trustlets is performed by means of a simple handshake protocol. The handshake requires that the initiator verifies the platform state, and that each party attests the other trustlet's state by checking the correctness of the relevant entries in the trustlet table and MPU registers. The initiator may additionally perform an integrity check of the responder's program code to ensure that it was not maliciously modified. After attesting each other, subsequent messages can be authenticated by means of a *cryptographic session token*.

5.11 TyTAN

TyTan [47] is an architecture for lightweight devices which provides isolation between tasks, secure IPC with sender and receiver authentication, and has real-time guarantees. Its TCB consists of both hardware and software components.

A core trusted hardware component of TyTAN is its Execution-Aware MPU (EA-MPU), which enforces program counter-based isolation. The EA-MPU ensures that each isolated task can only access its assigned memory regions. In addition, these tasks can only be invoked at a dedicated entry point.

Several static secure software components are part of the TCB. The *Secure Boot* task is invoked at boot, and is responsible for loading all other trusted software components. Each of these components is isolated from the rest of the system

by the EA-MPU. The EA-MPU supports loading and unloading of secure tasks at runtime by means of a driver. The *RoT for Measurement (RTM) task* is used to attest other tasks. This task calculates a cryptographic hash of the binary code of each created task, which serves as its identity id_t. The *Remote Attest* task uses a MAC to prove the authenticity of id_t to a remote verifier.

Secure IPC is done by means of the *IPC Proxy* task. This task is responsible for forwarding a message m from the sender S to the receiver R. For short messages, the sender invokes the proxy with the receiver's identity id_R and message m as parameters, which then copies m into R's memory. Since the EA-MPU ensures that only the proxy can write to R's memory, this implicitly authenticates m and id_S. For large messages, the proxy sets up a shared memory that is accessible only by the communicating tasks.

The *Secure Storage* task seals data by storing it encrypted in non-secure memory. It is encrypted with a task key that is derived from id_t. Tasks communicate with the secure storage via secure IPC. Finally, a trusted *Interrupt Multiplexor (Int Mux) task* is used to securely save the context of an interrupted task to its stack and clear the CPU registers before control is passed to the interrupt handler. Different interrupt handlers can be specified in the Interrupt Descriptor Table (IDT), which is protected by the EA-MPU.

TyTAN was implemented as an extension to Intel's Siskiyou Peak architecture, and uses the FreeRTOS real-time OS. The FreeRTOS preemptive scheduler was modified to support secure tasks. All secure software tasks were designed to be interruptible, or to have an upper bound on execution time. To support dynamic loading of tasks, FreeRTOS was extended with an ELF loader.

5.12 Sanctum

Sanctum [48] combines minimal hardware modifications with a trusted software component to offer an isolation scheme which is in many ways similar to SGX (section 5.8). Like SGX, Sanctum only allows enclaves to run at user level. Unlike it, physical attacks are not addressed by Sanctum, e.g., attacks on DRAM cannot be prevented because there is no Memory Encryption Engine (MEE) to encrypt code or data before being written out to memory.

In Sanctum, each enclave controls and manages its own page tables and handles its own page faults, whereas the former are managed by the OS or hypervisor in SGX. Furthermore, Sanctum ensures that each enclave is assigned a separate DRAM region corresponding to distinct sets in the shared Last-Level Cache (LLC). These two measures allow Sanctum to protect enclaves against software side-channel attacks where a malicious application or OS tries to learn information from an enclave's memory access pattern. In SGX, a potentially malicious OS can observe the accesses of any enclave at page granularity by reading the page table's *dirty* and *accessed* bits. Additionally, enforcing distinct cache sets per enclave protects against cache timing attacks.

Instead of implementing trusted functionality in microcode like SGX does, Sanctum uses a trusted software component called the *security monitor*. When booting a Sanctum system, measurement code within ROM is executed and calculates a hash of the security monitor, which is included in all further measurements, before giving control to the monitor. The security monitor then provides an API for enclave management, e.g., for creating and destroying enclaves. It also manages transitions into and out of enclaves, i.e., special *monitor calls* need to be used to enter and exit an enclave. In case of an interrupt, the security monitor takes care of saving the enclave's current state. After handling the interrupt, however, the enclave is entered at its entry point, and has to restore its state on its own. The environment within an enclave is also restricted, so that enclaves need to be exited for system calls and I/O.

Sanctum modifies the MMU in such a way that there are two Page Table Base Registers (PTBRs), one for untrusted code and one for the currently running enclave. Only the security monitor is able to change the contents of these registers. Furthermore, the modified MMU ensures that only certain pages can be referenced by enclave page tables. In more detail, metadata indicating valid pages is saved during enclave creation, and checked against after each page table walk. The metadata cannot be changed from software after enclave creation, during which the security monitor checks for overlapping pages or other invalid mappings, and writes it accordingly. Although the initial mappings are created by the OS, and copied to the enclave's page tables, the enclave is able to verify them by inspecting its own page tables and aborting if necessary.

6 Comparison

This section provides a detailed comparison of all architectures discussed in this paper. Table 1 compares all of them with respect to the security properties and architectural features given in 4. In addition, it is indicated for each architecture whether it was published by academic researchers, if its source code is public, and what Instruction Set Architecture (ISA) it was based on.

Except for the TPM and SMART, which were specifically designed for attestation, all architectures provide some isolation mechanism, which protects applications from each other and even the OS. In general, lightweight architectures include program counter-based access control in the memory controller, verifying each access. A common approach is to use a set of boundary registers which indicate the memory regions for a pre-defined number of protected modules. Since they already include an MMU, complex architectures extend it to include access control. At cache line or page granularity, the isolation is much coarser here than it is for lightweight architectures, where each memory access is checked.

It is interesting to see how different architectures implement remote attestation. Some architectures add a simple attestation protocol in hardware, based on symmetric primitives. These are cheaper than asymmetric algorithms in terms of computation and resource requirements. For example, Sancus (section 5.6) uses an HMAC based on SPONGENT. However, other designs instead opt to move their attestation functionality to software, relying on hardware functionality to protect it from the rest of the system. In this case, local attestation is used to protect the on-chip communication with the application being attested. This approach is especially attractive for complex protocols, like the one from SGX (section 5.8), which is based on a group signature scheme.

The same approach can be followed for other components of the TCB, especially for features which are

TABLE 1
Overview of all Hardware-Based Trusted Computing Architectures Detailed in Section 5

Architecture	Security Properties							Architectural Features							Other		
	Isolation	Attestation	Sealing	Dynamic RoT	Code Confidentiality	Side-Channel Resistance[1]	Memory Protection[2]	Lightweight	Coprocessor	HW-Only TCB	Preemption	Dynamic Layout	Upgradeable TCB	Backwards Compatibility	Open-Source	Academic	Target ISA
AEGIS [46]	●	●	●	●	●	○	●	○	○	●	●	●	○	●	○	●	–
TPM [47]	○	●	●	○	●	–	◐	○	●	●	–	–	○	●	○	○	–
TXT [22]	●	●	●	●	●	●	◐	○	●	●	○	●	○	●	○	○	x86_64
TrustZone [1]	●	○	○	●	○	○	○	○	○	●	●	●	○	●	○	○	ARM
Bastion [9]	●	○	●	●	●	○	●	○	○	○	●	●	●	●	○	●	UltraSPARC
SMART [14]	○	●	○	●	○	○	○	●	○	○	–	–	○	●	○	●	AVR/MSP430
Sancus [39]	●	●	○	●	○	○	○	●	○	●	○	○	○	◐	●	●	MSP430
Soteria [21]	●	●	○	●	●	○	○	●	○	●	○	○	○	◐	●	●	MSP430
SecureBlue++ [49]	●	○	●	●	●	○	●	○	○	●	●	●	○	●	○	○	POWER
SGX [35]	●	●	●	●	●	○	●	○	○	○	●	●	●	●	○	○	x86_64
Iso-X [15]	●	●	○	●	○	○	●	○	○	○	●	●	●	●	○	●	OpenRISC
TrustLite [28]	●	●	○	○	○	○	○	●	○	○	●	●	●	●	○	●	Siskiyou Peak
TyTAN [8]	●	●	●	●	○	○	○	●	○	○	●	●	●	●	○	●	Siskiyou Peak
Sanctum [12]	●	●	●	●	●	●	○	○	○	○	●	●	●	●	●	●	RISC-V

● = Yes; ◐ = Partial; ○ = No; – = Not Applicable
[1]Resistance against software side-channel attacks targeting memory access patterns only.
[2]Protection from physical attacks, both passive (e.g., probing) and active (e.g., fault injection).

expensive to implement in hardware. It has the additional advantage that the TCB can be upgraded, since it is partially implemented in software. The only exception is SMART, where the SW is stored in non-programmable ROM. In contrast, having a HW-only TCB implies that it cannot be upgradeable. This software will be part of the design's TCB, though, requiring users to trust that a potential attacker has no way of modifying its operation. All architectures have at least part of their TCB implemented in hardware, which is assumed to be immutable by attackers. Any software component which is part of the TCB relies on these features. HW-only TCBs can generally give much stronger guarantees, because no part of the architecture is vulnerable to software-level attackers. However, if carefully designed and implemented, some of its components can be moved to software, speeding up development and increasing flexibility. TrustZone (section 5.3) is the only architecture where a large amount of software is part of the TCB, because its isolation mechanism only supports two domains, and therefore includes the secure world OS.

All isolation architectures have similar attacker models, and consequently protect against the same types of vulnerabilities. There are two high-level categories of software attacks: code injection and code reuse attacks. The isolation mechanism protects against the former, since an attacker outside the module can no longer modify its code. Furthermore, attestation enables detection of any changes to the module at the time the measurement is taken. An entry point prevents external adversaries from performing the latter (section 4.1). However, neither mechanism can secure against vulnerabilities found inside the module itself. Software side-channels are a third category, but only Sanctum (section 5.12) addresses a specific instance of this attack type.

The trust boundaries typically extend to the CPU package, but in some cases external memories and peripherals are also included. This is the case for the TPM, which is a coprocessor connected to a shared system bus. Any other components on the same bus are therefore part of the TCB. However, when external memories and peripherals are not included in the TCB, there is also protection against physical memory attacks. For example, SecureBlue++ (section 5.7) transparently encrypts and decrypts cache lines when they are evicted or fetched from memory. Therefore, attackers who probe the memory or snoop the bus cannot obtain sensitive information. The hardware also maintains an integrity tree of all entries, defending against active memory attacks.

All discussed architectures modify the processor architecture itself, except for the TPM (section 5.2). SECA [49] is another instance where the security mechanisms are integrated outside of the CPU package. Instead, this architecture enforces configured *security contexts* at the bus level through a Security Enforcement Module (SEM). This hardware component monitors the bus traffic and interrupts the CPU when violations are detected. Different contexts can be configured by a secure kernel running on the processor, and it can update the currently active context at any time. For example, a security context can specify access rights to a specific memory range, isolating that region.

Sancus (section 5.6) is an example of a cooperative architecture without preemption (section 4.2). Such designs often rely on static memory layouts where all applications are stored in pre-defined memory locations, so that they know

where to call each other. Since only one application is running at the same time, monopolizing all resources, no software side-channels exist in these architectures (section 4.1).

All included designs either give programmers the choice to integrate their security mechanisms, or enable them transparently. Both approaches result in fully backwards compatible architectures, and with dynamic loading, the original binaries can even be used. Since a static layout requires recompilation, such architectures were listed as partially compatible. However, legacy applications remain just as vulnerable as before in most designs. TrustLite (section 5.10) is one exception, as it always provides isolation.

The goal of isolation is to protect modules from any other software running on the system. However, these components sometimes need to be able to communicate with each other, or even with untrusted applications, like the OS. This IPC is typically implemented in two ways. The fastest is to use processor registers for passing smaller messages. Larger messages are sent through shared memory regions. Some architectures even support secure shared memory, where modules can selectively allow others to access a memory region (e.g., SecureBlue++).

7 Conclusion

The goal of trusted computing is to protect applications and users from malicious software. It has increasingly gained interest in recent years, both from academia and industry, resulting in a variety of new mechanisms. We presented detailed descriptions of twelve hardware-based architectures, focusing on attestation and isolation designs, and compared them with respect to their security properties and architectural features. Our comparison shows that all architectures offer strong guarantees, but very few support all possible trusted computing mechanisms. The main differences are the size of the TCB, which sometimes contains software, and where the trust boundaries extend to. Furthermore, not all architectures support certain architectural features.

This paper shows there has been a lot of work in this area, but researchers and application developers still do not have widespread access to these technologies. Industry has only recently started building products which include them, and academic researchers rarely open-source their results, making it harder to extend their work. Therefore, there is still room for improvement, not only for attestation and isolation, but also for other trusted computing technologies.

Acknowledgments

We would like to thank Raoul Strackx for his valuable feedback, as well as the anonymous reviewers. This work was supported in part by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89), the KU Leuven Research Council through C16/15/058, the Flemish Government through FWO G.0130.13N and FWO G.0876.14N, and ERC Advanced Grant 695305. Pieter Maene is supported by a doctoral grant of the Research Foundation-Flanders (FWO).

References

[1] Proofpoint, "Proofpoint uncovers internet of things (IoT) cyberattack," Press Release, Jan. 2014.

[2] C. Miller and C. Valasek, "Remote exploitation of an unaltered passenger vehicle," presented at the Black Hat, Las Vegas, NV, USA, 2015.

[3] N. Falliere, L. O. Murchu, and E. Chien, "W32.Stuxnet dossier," Symantec, 2011.

[4] A. Martin, "The ten page introduction to trusted computing," Univ. Oxford, Oxford, U.K., Tech. Rep. RR-08–11, 2008.

[5] D. Gollmann, "Why trust is bad for security," *Electron. Notes Theoretical Comput. Sci.*, vol. 157, no. 3, pp. 3–9, 2006.

[6] C. Mundie, P. de Vries, P. Haynes, and M. Corwine, "Trustworthy computing," Microsoft, 2002.

[7] S. Lipner, "The trustworthy computing security development lifecycle," in *Proc. 20th Conf. Comput. Secur. Appl.*, 2004, pp. 2–13.

[8] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla, "SWATT: SoftWare-based attestation for embedded devices," in *Proc. 25th IEEE Symp. Secur. Privacy*, 2004, pp. 272–282.

[9] L. Martignoni, R. Paleari, and D. Bruschi, "Conqueror: Tamper-proof code execution on legacy systems," in *Proc. 7th Conf. Detection Intrusions Malware Vulnerability Assessment*, 2010, pp. 21–40.

[10] R. Jayaram Masti, C. Marforio, and S. Capkun, "An architecture for concurrent execution of secure environments in clouds," in *Proc. 20th Workshop Cloud Comput. Secur. Workshop*, 2013, pp. 11–22.

[11] R. Strackx and F. Piessens, "Fides: Selectively hardening software application components against kernel-level or process-level malware," in *Proc. 19th Conf. Comput. Commun. Secur.*, 2012, pp. 2–13.

[12] N. Avonds, R. Strackx, P. Agten, and F. Piessens, "Salus: Non-hierarchical memory access rights to enforce the principle of least privilege," in *Proc. 9th Conf. Secur. Privacy Commun. Netw.*, 2013, pp. 252–269.

[13] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*, 1st ed. Boca Raton, FL, USA: CRC Press, 1996.

[14] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," in *Proc. 3rd Conf. Comput. Syst.*, 2008, pp. 315–328.

[15] R. Strackx, J. Noorman, I. Verbauwhede, B. Preneel, and F. Piessens, "Protected software module architectures," in *Proc. 13th Conf. Inf. Secur. Solutions*, 2013, pp. 241–251.

[16] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Trans. Inf. Theory*, vol. IT-29, no. 2, pp. 198–208, Mar. 1983.

[17] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. 14th Conf. Comput. Commun. Secur.*, 2007, pp. 552–561.

[18] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: Architecture for tamper-evident and tamper-resistant processing," in *Proc. 17th Annu. Int. Conf. Supercomputing*, 2003, pp. 160–171.

[19] *TPM Main: Part 1 Design Principles, Version 1.2, Revision 116 ed., Trusted Computing Group*, Beaverton, OR, USA, 2011.

[20] K. Kursawe, D. Schellekens, and B. Preneel, "Analyzing trusted platform communication," in *Proc. ECRYPT Workshop CRyptographic Advances Secure Hardware*, 2005, pp. 1–8.

[21] *Trusted Platform Module Library: Part 1: Architecture, Family 2.0, Level 00, Revision 01.16 ed., Trusted Computing Group*, Beaverton, OR, USA, 2014.

[22] D. Grawrock, *Dynamics of a Trusted Platform: A Building Block Approach*, 1st ed. Santa Clara, CA, USA: Intel Press, 2009.

[23] J. M. McCune, et al., "TrustVisor: Efficient TCB reduction and attestation," in *Proc. 31st IEEE Symp. Secur. Privacy*, 2010, pp. 143–158.

[24] *GlobalPlatform Device Technology TEE Client API Specification, Revision 0.17 ed., GlobalPlatform*, Redwood City, CA, USA, 2010.

[25] *GlobalPlatform Device Technology TEE Internal API Specification, Revision 1.0 ed., GlobalPlatform*, Redwood City, CA, USA, 2011.

[26] "Security technology building a secure system using TrustZone technology," ARM, Tech. Rep. PRD29-GENC-009492C, 2009.

[27] D. Champagne and R. B. Lee, "Scalable architectural support for trusted software," in *Proc. 16th Conf. High-Performance Comput. Archit.*, 2010, pp. 1–12.

[28] K. El Defrawy, A. Francillon, D. Perito, and G. Tsudik, "SMART: Secure and minimal architecture for (establishing a dynamic) root of trust," in *Proc. 19th Symp. Netw. Distrib. Syst. Secur.*, 2012, pp. 1–15.

[29] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik, "A minimalist approach to remote attestation," in *Proc. 17th Conf. Des. Autom. Test Europe*, 2014, Art. no. 244.

[30] N. Asokan, et al., "SEDA: Scalable embedded device attestation," in *Proc. 22nd Conf. Comput. Commun. Secur.*, 2015, pp. 964–975.

[31] J. Noorman, et al., "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base," in *Proc. 22nd USENIX Conf. Secur.*, 2013, pp. 479–494.
[32] A. Bogdanov, M. Knežević, G. Leander, D. Toz, K. Varıcı, and I. Verbauwhede, "SPONGENT: A lightweight hash function," in *Proc. 13th Workshop Cryptographic Hardware Embedded Syst.*, 2011, pp. 312–325.
[33] J. Götzfried, T. Müller, R. de Clercq, P. Maene, F. Freiling, and I. Verbauwhede, "Soteria: Offline software protection within low-cost embedded devices," in *Proc. 31st Conf. Comput. Secur. Appl.*, 2015, pp. 241–250.
[34] P. Williams and R. Boivie, "CPU support for secure executables," in *Proc. 4th Conf. Trust Trustworthy Comput.*, 2011, pp. 172–187.
[35] R. Boivie and P. Williams, "SecureBlue++: CPU support for secure executables," Yorktown Heights, NY, USA: IBM, Tech. Rep. RC25287, 2013.
[36] F. McKeen, et al., "Innovative instructions and software model for isolated execution," in *Proc. 2nd Workshop Hardware Archit. Support Secur. Privacy*, 2013, Art. no. 10.
[37] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. del Cuvillo, "Using innovative instructions to create trustworthy software solutions," in *Proc. 2nd Workshop Hardware Archit. Support Secur. Privacy*, 2013, Art. no. 11.
[38] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative technology for CPU based attestation and sealing," in *Proc. 2nd Workshop Hardware Archit. Support Secur. Privacy*, 2013, pp. 1–6.
[39] Intel, "Intel software guard extensions programming reference (329298–002US)," Santa Clara, CA, USA, 2014.
[40] S. P. Johnson, V. R. Scarlata, C. V. Rozas, E. Brickell, and F. McKeen, "Intel SGX: EPID provisioning and attestation services," Intel, 2016.
[41] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.
[42] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with Haven," in *Proc. 11th USENIX Symp. Operating Syst. Des. Implementation*, 2014, pp. 267–283.
[43] F. Schuster, et al., "VC3: Trustworthy data analytics in the cloud using SGX," in *Proc. 36th IEEE Symp. Secur. Privacy*, 2015, pp. 38–54.
[44] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," AMD, 2016.
[45] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley, "Iso-X: A flexible architecture for hardware-managed isolated execution," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2014, pp. 190–202.
[46] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A security architecture for tiny embedded devices," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, Art. no. 10.
[47] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "TyTAN: Tiny trust anchor for tiny devices," in *Proc. 52nd Conf. Des. Autom.*, 2015, pp. 1–6.
[48] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," *IACR Cryptology ePrint Archive*, vol. 2015, no. 564, pp. 1–22, 2015.
[49] J. Coburn, S. Ravi, A. Raghunathan, and S. Chakradhar, "SECA: Security-enhanced communication architecture," in *Proc. 8th Conf. Compilers Archit. Synthesis Embedded Syst.*, 2005, pp. 78–89.

Pieter Maene is a research assistant with COSIC Research Group, KU Leuven. His research interests include trusted computing architectures, hardware-software co-design, and hardware implementations of cryptographic algorithms.

Johannes Götzfried is a research assistant at the chair for IT Security Infrastructures, Friedrich-Alexander-Universität (FAU), Erlangen-Nürnberg. His research interests include trusted computing, system security, and physical security.

Ruan de Clercq is a research assistant with COSIC Research Group, KU Leuven. His research interests include embedded security, computer security architectures, and applied cryptography.

Tilo Müller is a post-doctoral researcher at the chair for IT Security Infrastructures, Friedrich-Alexander-Universität (FAU), Erlangen-Nürnberg. His research interests include system security, mobile security, and software protection.

Felix Freiling is professor of computer science with Friedrich-Alexander-Universität (FAU), Erlangen-Nürnberg. His research interests cover theory and practice of dependable systems.

Ingrid Verbauwhede is a professor of electrical engineering with KU Leuven. Her main interests include the design and the design methods for secure embedded circuits and systems. She is a fellow of the IEEE.