SGXIO: Generic Trusted I/O Path for Intel SGX

Samuel Weiser
Graz University of Technology, Austria
samuel.weiser@iaik.tugraz.at

Mario Werner
Graz University of Technology, Austria
mario.werner@iaik.tugraz.at

ABSTRACT

Application security traditionally strongly relies upon security of the underlying operating system. However, operating systems often fall victim to software attacks, compromising security of applications as well. To overcome this dependency, Intel SGX allows to protect application code against a subverted or malicious OS by running it in a hardware-protected enclave. However, SGX lacks support for generic trusted I/O paths to protect user input and output between enclaves and I/O devices.

This work presents SGXIO, a generic trusted path architecture for SGX, allowing user applications to run securely on top of an untrusted OS, while at the same time supporting trusted paths to generic I/O devices. To achieve this, SGXIO combines the benefits of SGX's easy programming model with traditional hypervisor-based trusted path architectures. Moreover, SGXIO can tweak insecure debug enclaves to behave like secure production enclaves. SGXIO surpasses traditional use cases in cloud computing and digital rights management and makes SGX technology usable for protecting user-centric, local applications against kernel-level keyloggers and likewise. It is compatible to unmodified operating systems and works on a modern commodity notebook out of the box. Hence, SGXIO is particularly promising for the broad x86 community to which SGX is readily available.

1. INTRODUCTION

Software vulnerabilities are still a predominant issue for application security. A major reason for this fact is code complexity, which makes traditional secure design paradigms like software verification or testing reach its limits. Hence, research focuses on securing sensitive code only and executing it in architecturally isolated containers, often referred to as enclaves. Since an enclave is protected against all non-enclave code, the whole OS stack can safely be considered untrusted.

Intel SGX [14] provides such enclaves in its latest x86 mainline CPUs. It targets high-performance cloud computing, where the cloud provider is entirely distrusted, as well as Digital Rights Management (DRM). In order to protect not only enclave execution but also user I/O, one requires trusted paths between enclaves and I/O devices. Currently, SGX only works with proprietary trusted paths like Intel Protected Audio Video Path (PAVP), which rely on the Intel Management Engine (ME) [13,33]. However, proprietary trusted paths are hard to analyze regarding security. Moreover, they are not generic and address specific devices and scenarios only. Unfortunately, SGX lacks support for generic trusted paths.

Contributions. In this work we present the concept of SGXIO, which is, to our knowledge, the first generic trusted path architecture for SGX. SGXIO protects secure user applications as well as associated trusted paths against an untrusted OS. User applications benefit from SGX protection while trusted paths are established via a small and trusted hypervisor. To that end, we identify and solve several challenges in linking the security domains of SGX and the trusted hypervisor. This allows a remote party to attest not only enclave code but also the whole trusted path setup. Also, SGXIO allows human end users to verify trusted paths without requiring additional hardware. SGXIO improves upon existing generic trusted paths for x86 systems with an easier and more intuitive programming model. Furthermore, we show how SGXIO can tweak debug enclaves to behave like production enclaves. Therefore, the trusted hypervisor selectively disables enclave debug instructions. Finally, we give a novel zero-overhead, non-interactive key transport scheme for establishing a 128-bit symmetric key between two local SGX enclaves. The extended version of this paper can be found in [39].

The rest of this paper is structured as follows: Section 2 gives related work, followed by an overview of SGX in Section 3. Section 4 discusses the threat model and challenges. Section 5 presents our SGXIO architecture while Section 6 gives a thorough security analysis. Section 7 shows how SGXIO can apply the debug enclave tweak. It is followed by a conclusion in Section 8.

2. RELATED WORK

This section discusses prior work on isolated execution and trusted paths. Specifically, we compare to ARM TrustZone, which is ARM's counterpart to Intel SGX.

2.1 Isolated Execution

There exist various techniques for isolated execution, which range from pure hypervisor designs [7] over hardware-software co-designs [6,8,34] to pure hardware extensions [3,5,9,10,14,

CODASPY'17, March 22 – 24, 2017, Scottsdale, AZ, USA

ISBN 978-1-4503-4523-1/17/03…$15.00

DOI: http://dx.doi.org/10.1145/3029806.3029822

29]. SGXIO uses Intel SGX [14], which, from a functional perspective, cumulates previous work in software isolation, attestation and transparent memory encryption.

The Trusted Platform Module (TPM) [36] is a security co-processor which does not directly offer software isolation on its own. However, it can be used in conjunction with Intel TXT [16] to set up an isolated execution environment [25].

2.2 Trusted Paths

There exist various attempts for integrating trusted paths directly into existing commodity OSes. However, they usually suffer from a bloated TCB, covering the whole OS [11,22, 24,37]. More sound trusted paths consider the OS untrusted. One can distinguish between generic trusted paths and specific trusted paths. Latter are limited to specific devices or scenarios.

Generic Trusted Paths. Zhou *et al.* establish a generic trusted path in a pure hypervisor-based design [42]. They show the first comprehensive approach on x86 systems, protecting a trusted path all the way from the application level down to the device level. They consider PCI device misconfiguration, DMA attacks as well as interrupt spoofing attacks. However, pure hypervisor-based designs come at a price. They strictly separate the untrusted stack from the trusted one. Hence, the hypervisor is in charge of managing all secure applications and all associated resources itself. This includes secure process and memory management with scheduling, verified launch and attestation. In contrast, SGXIO uses the comparably easy programming model of SGX enclaves, in which the untrusted OS is in charge of managing secure enclaves. Moreover, SGX provides verified launch and attestation out of the box.

Specific Trusted Paths. In [41] and [43], Zhou *et al.* discuss a trusted path to a single USB device. Yu *et al.* show how to apply the trusted path concept to GPU separation [40]. Filyanov *et al.* discuss a pure uni-directional trusted path [12]. However, their approach is limited to trusted input paths only and the OS is suspended during secure input. In contrast, SGXIO supports full parallelism between the untrusted OS and secure applications.

Others use dedicated I/O devices, which natively support encryption [31,32]. This allows applications to open a cryptographic channel to it, bypassing any untrusted software. However, this concept does not generalize to arbitrary I/O devices, especially legacy devices.

Many trusted paths build on proprietary hardware and software like Intel's Protected Audio Video Path (PAVP) as well as its successor, Intel Insider [21,33]. Both rely on Intel's proprietary Management Engine (ME), which hinders transparent security assessment. Hoekstra *et al.* outline integration of PAVP in SGX applications to achieve secure video conferencing and one-time password generation [13]. However, they do not come up with any trusted input path. In [33], Ruan describes a Protected Transaction Display (PTD) application, which uses PAVP to securely obtain a one-time PINs from the user.

2.3 ARM TrustZone

ARM TrustZone combines secure execution with trusted path support. A TrustZone compatible CPU provides a secure world mode, which is orthogonal to classical privilege levels. The secure world is isolated against the normal world and operates a whole trusted stack, including security kernel, device drivers and applications. In addition, TrustZone is complemented by a set of hardware modules, which allow strong isolation of physical memory as well as peripherals. Also, device interrupts can be directly routed into the secure world. TrustZone can be combined with a System MMU, similar to an IOMMU, which can prevent DMA attacks. Thus, TrustZone not only allows isolated execution [35] but also generic trusted paths [23], which is a significant advantage over SGX. In contrast to SGX, TrustZone does not distinguish between different secure application processes in hardware. It requires a security kernel for secure process isolation, management, attestation and similar.

3. SOFTWARE GUARD EXTENSIONS

Intel Software Guard Extensions (SGX) define a new set of x86 instructions for enclaves [2,13–15,17,18,26]. User applications can host enclaves in their virtual address spaces. Any access into enclave memory is prohibited by the CPU, while enclaves can access their hosting application's memory for data exchange. In addition, SGX encrypts enclave memory when written to DRAM. The OS is entirely distrusted and is supervised by the CPU in all enclave management operations. The Trusted Computing Base (TCB) only consists of enclave code and the CPU itself.

SGX implements verified enclave launch. During enclave initialization the CPU hashes enclave memory into a protected register, called `MRENCLAVE`. Before running the enclave, the CPU verifies `MRENCLAVE` against a vendor-signed version. Hence, `MRENCLAVE` vouches for integrity of the enclave startup. Other parties can verify `MRENCLAVE` via local or remote attestation. Attestation is based on a report structure holding `MRENCLAVE`, which is cryptographically signed by the CPU. The report structure is able to hold additional user data, which can be used to authentically exchange information between enclaves to agree on an encryption key, for example. Moreover, SGX can derive per-enclave sealing keys from their `MRENCLAVE` values to allow encrypted storage of secrets [1].

SGX distinguishes between debug and production enclaves. Former can be accessed by the OS via SGX debug instructions while latter cannot. During enclave initialization, developers choose between debug and production mode. This choice yields different `MRENCLAVE` values for either option, making a debug enclave distinguishable from a production enclave.

SGX restricts enclave execution via a launch enclave, which is discussed controversially [4]. All production enclaves need to be licensed by Intel. Unlicensed enclaves can only be launched in debug mode [19].

4. THREAT MODEL AND CHALLENGES

SGXIO utilizes SGX as building block to provide isolated execution. However, the threat models of pure SGX and SGXIO differ. This section elaborates on the differences as well as the challenges arising from the combination of SGX with a trusted hypervisor.

4.1 Distinction

SGX enforces a minimal TCB, covering only the processor and enclave code. All other software components (e.g., OS, hypervisor) as well as the processor's physical environment is considered potentially malicious. Therefore, SGX not only considers logical attacks but also physical attacks. This threat model perfectly fits the requirements of secure cloud

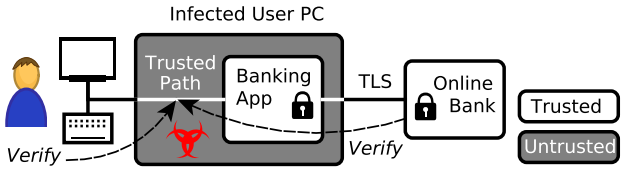

Figure 1: SGXIO enhances security of online banking via trusted paths.

computing in which a customer wants to protect enclaves against an untrusted cloud provider. Also, content providers can use SGX to enforce a DRM scheme on an untrusted consumer PC.

In a local setting, however, a user wants to benefit from SGX by protecting user-centric applications against a potentially compromised OS. Although user apps can be protected by SGX enclaves, the communication between user apps and the user via I/O peripherals cannot. A compromised OS can easily intercept and alter I/O communication. This local setting overcharges plain SGX as it somehow contradicts its threat model, which considers the physical environment, and therefore also the local user, a threat.

Currently, in order to achieve a trusted path with SGX, one has to rely on encrypted interfaces like PAVP. However, the prevalence of unencrypted I/O devices in todays computers and the lack of support to securely communicate with these devices demands other, more generic mechanisms.

SGXIO fixes this shortcoming by extending SGX with a generic trusted path, which protects user I/O against the OS. Additionally, SGXIO provides attestation mechanisms to verify that trusted paths are established and functional.

SGXIO supports user-centric applications like confidential document viewers, login prompts, password safes, secure conferencing and secure online banking. To take latter as example, with SGXIO an online bank cannot only communicate with the user via TLS but also with the end user via trusted paths between banking app and I/O devices, as depicted in Figure 1. This means that sensitive information like login credentials, the account balance, or the transaction amount can be protected even if other software running on the user's computer, including the OS, is infected by malware.

Having a trusted path has implications on the threat model of SGXIO. A physical attacker has direct access to I/O devices and can impersonate the user without subverting trusted paths. Thus, trusted paths can only protect against logical attacks but cannot provide physical security at all. The following section explains the threat model of SGXIO in detail.

4.2 Threat Model

In general, adversaries attempt to subvert a trusted path between a user app and an I/O device. They succeed when breaking the confidentiality or authenticity of a trusted path.

Logical attacks are the main concern of SGXIO. Attackers are assumed to have full control over the OS and know the whole software configuration, including all enclave code. This is a realistic scenario, addressing both local and remote attacks, which might yield kernel privileges to attackers. Attackers can therefore directly attack enclave interfaces visible to the OS by running enclaves in a fake environment within the OS. Also, attackers can dynamically load and execute custom user apps and drivers and open other trusted path sessions. Moreover, attackers can tamper with the boot image of the trusted hypervisor, which SGXIO is based on.

Indirect attacks on a trusted path can be performed by misconfiguring devices under OS control to interfere with a trusted path, as outlined by Zhou *et al.* [42]. Also, malicious Direct Memory Access (DMA) requests could be issued and interrupts could be spoofed.

All code in the trusted computing base (e.g., secure user applications, secure I/O drivers and the trusted hypervisor) is assumed to be correct and not vulnerable to logical attacks. Using a formally verified hypervisor such as seL4 [20] supports this assumption.

Physical attacks are not considered in SGXIO, as already explained, since the user interacting with the system has to be trusted anyway. As with SGX, Denial-of-Service (DoS) and side channel attacks are out of scope for SGXIO.

Note that SGXIO requires a modern Intel platform with SGX support as well as support for TPM-based trusted boot. All hardware (CPU, chipset, peripherals) is expected to work correctly.

4.3 Challenges

SGXIO combines SGX with a trusted hypervisor to provide a generic trusted path. However, the hypervisor and SGX form two disjoint security domains with two different trust anchors, which are not designed to collaborate. Subsequently, connecting both domains is a non-trivial task.

This essentially breaks down to two major challenges which had to be solved: First, the security domains of the hypervisor and SGX enclaves have to be linked. More concretely, we need a way for SGX enclaves to check the presence and the authenticity of the hypervisor. We name this problem *hypervisor attestation*. Once the hypervisor is attested, it extends trust to any trusted path it establishes.

Second, the SGXIO architecture relies on multiple SGX enclaves, which are executed in different security contexts (trusted hypervisor vs. untrusted OS). However, SGX enclaves are unaware of their context, making them vulnerable to *enclave virtualization attacks*. SGXIO prevents such attacks via a careful interface design between both contexts.

5. SGXIO ARCHITECTURE

This section presents our SGXIO architecture and elaborates on its isolation guarantees. We discuss the design of secure user apps, secure I/O drivers as well as the hypervisor.

5.1 Architecture

SGXIO is composed of two parts: a trusted stack and a Virtual Machine (VM), as seen in Figure 2. The trusted stack contains a small security hypervisor, one or more secure I/O drivers, which we simply call *drivers*, as well as a Trusted Boot (TB) enclave. The VM hosts an untrusted commodity OS like Linux, which runs secure user applications, also abbreviated with *user apps*.

User apps want to communicate securely with the end user. They open an encrypted communication channel to a secure I/O driver to tunnel through the untrusted OS. The driver in turn requires secure communication with a generic user I/O device, which we term *user device*. To achieve this, the hypervisor exclusively binds user devices to the corresponding

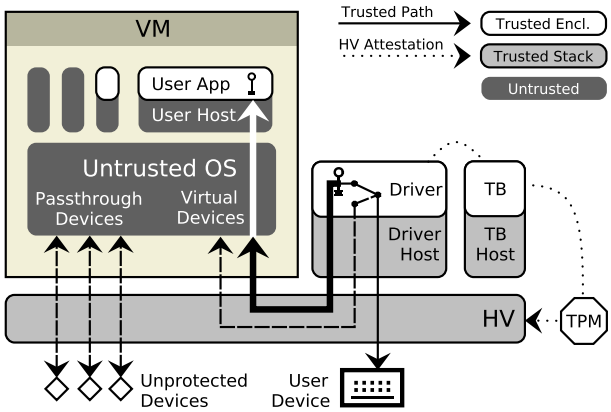

Figure 2: The trusted stack consists of a hypervisor (HV), a Trusted Boot (TB) enclave and one or more secure I/O drivers. The Virtual Machine (VM) operates an untrusted OS, hosting secure user apps. To establish a trusted path (solid line), the driver encrypts secret data (bold line) for a user app. The TB enclave allows drivers to attest the HV.

drivers. Note that any other device is directly assigned to the VM. I/O on those unprotected devices directly passes through the hypervisor without performance penalty. The trusted path names both, the encrypted user-app-to-driver communication and the exclusive driver-to-device binding. It is indicated with a solid line in Figure 2. Drivers use the TB enclave to get assured of correct trusted path setup by attesting the hypervisor, which is indicated by a dotted line.

5.2 Isolation Guarantees

SGXIO establishes a trusted path all the way from a user app to the user device. This requires isolation on several layers. First, all trusted stack memory needs to be isolated from the untrusted OS and Direct Memory Access (DMA). Second, the trusted path itself requires isolation from the OS. Third, the user device needs isolation from all other devices which are under control of the OS. This section outlines how SGXIO meets these isolation requirements.

Trusted Memory Isolation is a prerequisite for securely executing trusted code in an untrusted environment. This affects user apps as well as the trusted stack. To achieve memory isolation of the user app, it is executed within an enclave. SGX isolates all enclave memory from the untrusted OS. To achieve memory isolation for the trusted stack, the hypervisor confines the untrusted OS in a VM. Moreover, the hypervisor implements a strict memory partitioning by configuring the Memory Management Unit (MMU) appropriately. This prevents the OS from escaping the VM and tampering with the trusted stack.

Direct Memory Access (DMA) is a more subtle threat to memory isolation [42]. A DMA-capable device can directly access memory, bypassing any MMU protection and potentially violating integrity and confidentiality of trusted memory. SGX prevents DMA from accessing enclave memory, hence the user app is safe against DMA attacks [14]. Likewise, the trusted stack has to be protected against such attacks. Modern chipsets typically incorporate an I/O Memory Management Unit (IOMMU), also termed VT-d on Intel systems. The IOMMU restricts device DMA to specific portions of RAM. By properly configuring the IOMMU, the hypervisor can protect the whole trusted stack against device DMA attacks.

Trusted Path Isolation. The trusted path has to be protected on two layers, namely the communication between user app and driver as well as the interaction between driver and user device. The user app communicates with the driver via the untrusted OS stack, hence encryption is necessary. The interaction between driver and user device is protected by the hypervisor. Therefore, the hypervisor establishes an exclusive binding between a driver and the corresponding user device. Moreover, the hypervisor mutually isolates all drivers. Thus, an attacker, loading arbitrary driver code at will, cannot interfere with trusted paths established by other drivers.

User Device Isolation. As outlined before, a malicious OS could misconfigure devices to interfere with the trusted path. In that way, OS-controlled PCI devices could be forced to overlap their MMIO region or I/O port range with those of the user device or issue forged interrupts on behalf of the user device. To protect against these attacks, Zhou *et al.* implement several policies in the hypervisor to detect and prevent malicious device configurations [42]. This effectively isolates user devices from other OS-controlled devices. Their approach is also applicable to SGXIO.

5.3 User App Design

Secure user apps process all sensitive data inside an enclave. Sensitive data leaves an enclave only in an encrypted form. For example, the user enclave can securely communicate with the driver enclave or a remote server via encrypted channels or store sensitive data offline using SGX sealing.

To set up an encrypted channel between enclaves, one needs to share an encryption key. Anati *et al.* propose Diffie-Hellman over SGX local attestation for this purpose [1,18]. However, local attestation inherently provides a much faster way of exchanging key material. We give a novel, lightweight key transport scheme, which comes with just a single unidirectional invocation of local attestation. We reuse the already pre-shared report key to derive random 128-bit encryption keys. The scheme works as follows: The user enclave generates a local attestation report over a random salt, targeted at the driver enclave. However, instead of delivering the actual report to the driver enclave, the user enclave keeps it private and uses the report's MAC as symmetric key. It then sends the salt and its identity to the driver enclave, which can recompute the MAC to obtain the same key. Details of this scheme are given in [39].

5.4 Driver Design

Drivers are hosted and protected by the hypervisor. Although hypervisor protection is sufficient to isolate drivers from the untrusted OS, actual driver logic is in addition executed in an enclave. This helps in setting up an encrypted communication channel with user apps, as previously described. Also, driver enclaves are subject to attestation, allowing identification via their `MRENCLAVE` values.

As shown in Figure 2, drivers can multiplex I/O data between user app and OS to account for trusted path setup and tear down, making SGXIO compatible to legacy OSes.

5.5 Hypervisor Design

The hypervisor enforces a bunch of isolation guarantees,

as previously outlined: First, it isolates all trusted stack memory. Second, it binds a user device exclusively to the corresponding driver and mutually isolates drivers. This achieves trusted path isolation. Third, it isolates user devices from malicious interference with other devices.

We recommend to use seL4 as hypervisor, as it directly supports isolation of trusted memory as well as user device binding via its capability system [28]. With seL4, isolation breaks down to a correct distribution of memory and device capabilities among the VM and the drivers.

seL4 makes heavy use of Intel's VT-x hardware virtualization to intercept illegitimate memory or device accesses. Thus, VT-x also helps in blocking device misconfiguration attacks [42] and achieving user device isolation. Furthermore, seL4 uses Intel VT-d, also referred to as IOMMU, to protect against DMA attacks from misconfigured devices. Moreover, seL4 is formally verified [20,27], which helps making strong claims about security of the trusted path.

6. DOMAIN BINDING AND ATTESTATION

Having explained our SGXIO architecture, this section elaborates on challenges which arise when binding the SGX domain with the trusted hypervisor domain. Specifically, this covers trusted boot and hypervisor attestation. We discuss how to protect hypervisor attestation against remote TPM attacks as well as enclave virtualization attacks. Having a domain binding in place allows remote attestation of trusted paths as well as user verification.

6.1 Challenges

SGXIO enables a remote party as well as a local user to verify security of trusted paths. In the first place, this requires a domain binding between SGX and the trusted hypervisor. In the second place, an appropriate user verification mechanism needs to be in place which is both secure and usable.

Domain Binding. In order to bind the SGX and the hypervisor domain, the hypervisor must level up to certain security guarantees SGX regarding isolated code execution. In SGX, all enclave memory is isolated from the rest. Moreover enclave loading is guarded by a verified launch mechanism, which can be attested to other parties. SGXIO rebuilds similar mechanisms for the hypervisor. Isolation of trusted memory has already been discussed in Section 5.2. Verified launch is implemented via *trusted boot* of the hypervisor with support for *hypervisor attestation*.

With trusted boot and hypervisor attestation in place, SGXIO can bind the SGX and the hypervisor domain. The binding needs to be bidirectional, allowing both the hypervisor and SGX enclaves to put trust in the other domain. One direction is easy: The hypervisor can extend trust to SGX by running enclaves in a safe, hypervisor-protected environment. These enclaves can in turn use local attestation to extend trust to any other enclaves in the system. However, the opposite direction is challenging: On the one hand, enclaves need confidence that the hypervisor is not compromised and binds user devices correctly to drivers. Effectively, this requires enclaves to invoke hypervisor attestation. SGXIO achieves this with assistance of the TB enclave. On the other hand, SGX is not designed to cooperate with a trusted hypervisor. Recall that SGX considers all non-enclave code untrusted. In fact, SGX explicitly prohibits use of any instruction inside an enclave that might be used to communicate with the hypervisor [17]. Even if hypervisor attestation succeeds, an enclave cannot easily learn whether it is legitimately executed by the hypervisor or virtualized by an attacker in a fake environment. This makes driver enclaves and the TB enclave vulnerable to virtualization attacks. SGXIO defends against such attacks by hiding hypervisor attestation from the untrusted OS.

User Verification. An end user wants to be able to verify if he is indeed communicating with the correct user app via a trusted path. This is non-trivial because the user cannot simply evaluate an cryptographic attestation report. Instead, the user requires some form of notification whether a trusted path is present. This notification needs to be unforgeable to prevent the OS from faking it. Moreover, it needs to help the user distinguish different user apps, not least because an attacker might also run arbitrary user apps under his control.

6.2 Trusted Boot & Hypervisor Attestation

Trusted boot does not prohibit booting a compromised hypervisor but uncovers any such compromise. To achieve this, trusted boot makes use of a TPM to measure the whole boot process, starting from a trusted piece of firmware code up to the hypervisor image. Each boot stage measures the next one in a cryptographic log inside the TPM using the `extend` operation. All measurements are cumulated in a TPM Platform Configuration Register (PCR). The final PCR value reflects the whole boot process and is used to prove integrity of the hypervisor to other parties.

Hypervisor Attestation allows enclaves to verify the trusted boot process in order to get assured of hypervisor's integrity. Since the hypervisor is responsible for loading drivers and doing trusted path setup, its attestation also vouches for security of all trusted paths.

To ease hypervisor attestation, SGXIO uses a Trusted Boot (TB) enclave which attests the hypervisor once. Afterwards, any driver enclave can query the TB enclave to get approval if hypervisor attestation succeeded, see Figure 3/I. The driver enclaves in turn can communicate the attestation result to user apps, which can finally implement a mechanism for remote parties or the end user to verify a trusted path.

To attest the hypervisor, the TB enclave needs to verify the PCR value, obtained during trusted boot. Therefore, the TB enclave requests a TPM `quote` [36], which contains a cryptographic signature over the PCR value alongside with a fresh nonce. This ensures not only integrity of the PCR value but also prevents replay attacks.

6.3 Attacks

The interaction between TB enclave and TPM is crucial for security of the hypervisor attestation scheme. One has to prevent remote TPM attacks as well as enclave virtualization attacks, which is outlined in the following.

6.3.1 Remote TPM attacks.

Naive hypervisor attestation is vulnerable to remote TPM attacks, also called cuckoo attacks [30], as shown in Figure 3/II. Here, the attacker compromises the hypervisor image, which yields a wrong PCR value during trusted boot. To avoid being detected by the TB enclave, the attacker generates a valid `quote` on an attacker-controlled remote TPM and feeds it into the TB enclave, which successfully approves the compromised hypervisor.

Defense. In order to stop cuckoo attacks, the TB enclave needs to identify the TPM it is talking to, *e.g.,* by means of

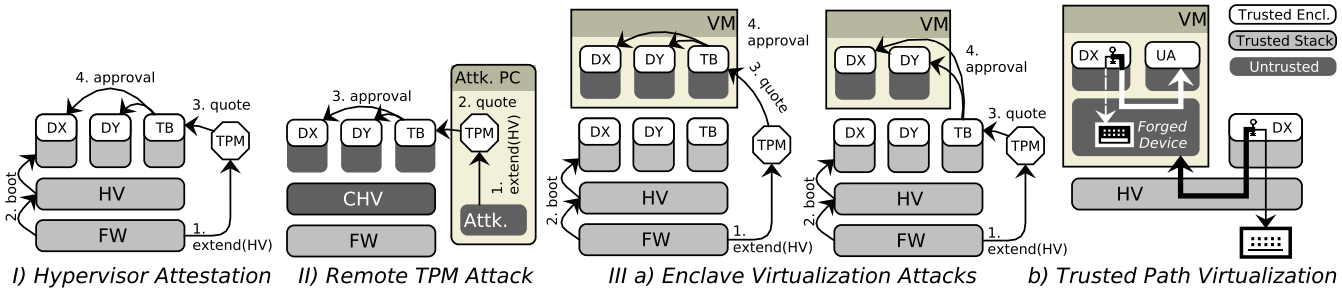

Figure 3: I) During trusted boot, Firmware (FW) measures the Hypervisor (HV) via a TPM. The TB enclave attests the HV via a TPM `quote` and approves other drivers (DX) and (DY). II) An attacker injects a remotely-generated `quote` to hide the presence of a Compromised HV (CHV). IIIa) An attacker diverts steps 3 or 4 to a virtualized environment. IIIb) This allows to virtualize a trusted path to a user app (UA).

the TPM's Attestation Identity Key (AIK) used for signing the `quote`. This allows the TB enclave to verify the origin of the `quote`. To make the AIK known to the TB enclave, one has to provision it, e.g., during initial system integration. The TB enclave stores the provisioned AIK by means of SGX sealing. Since sealing uses a CPU-specific encryption key, an attacker cannot trick the TB enclave to unseal an AIK not sealed by the same CPU. This effectively binds execution of the TB enclave to the TPM.

Provisioning of AIKs could be done by system integrators. One has to introduce proper measures to prevent attackers from provisioning arbitrary AIKs. For example, the TB enclave could encode a list of public keys of approved system integrators, which are allowed to provision AIKs.

6.3.2 *Enclave Virtualization Attacks*

In an enclave virtualization attack, the attacker does not compromise the actual trusted boot process. Rather, he virtualizes driver enclaves or even the TB enclave in a fake environment on the same computer, as depicted in Figure 3/IIIa. To make hypervisor attestation for the virtualized enclaves succeed, the attacker diverts the legitimate TPM quote or the TB enclave approval to the virtualized TB enclave or driver enclaves, respectively. As shown in Figure 3/IIIb, the attacker can now impersonate the user by rerouting user apps to a virtualized driver, reading driver's output and providing fake input. Note that the attacker did not change enclave code. Hence, SGX will generate the same `MRENCLAVE` value and thus the same derived cryptographic keys for both, legitimate and virtualized enclave instances. Neither the TB enclave nor the driver enclave or a user app can detect such virtualization. However, the attacker does not learn actual user input, which still arrives at the legitimate driver enclave.

Defense. The problem of enclave virtualization stems from the design of SGX which treats all enclaves equally, regardless of the security context they are executed in. As a defense, SGXIO restricts the communication interface between the hypervisor and the OS context. Therefore, the hypervisor hides the TPM as well as the TB enclave from the untrusted OS[1]. Only the legitimate TB enclave is given access to the TPM. Thus, the TB enclave might only succeed in hypervisor attestation if it has been legitimately launched by the hypervisor. Likewise, only legitimate driver enclaves are granted access to the legitimate TB enclave by the hypervisor. A driver enclave might only get approval if it can talk to the legitimate TB enclave, which implies that the driver enclave too has been legitimately launched by the hypervisor.

[1]This breaks path 3 on the left side and path 4 on the right side of Figure 3/IIIa, respectively.

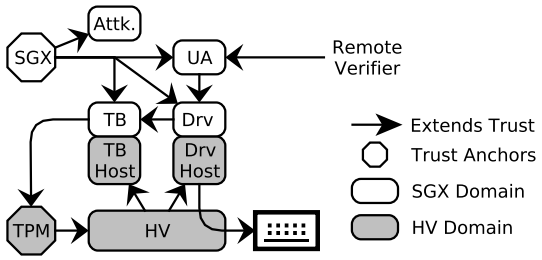

Figure 4: Trust hierarchy of SGXIO.

6.4 Remote Trusted Path Attestation

As already mentioned, hypervisor attestation vouches for security of trusted paths and serves as basis for remote attestation. This section describes the whole trust hierarchy involved in remote attestation, as shown in Figure 4.

SGXIO has two main hardware trust anchors, namely SGX and the TPM. SGX extends trust to all enclaves running on the system by means of verified launch. This also includes enclaves with attacker-controlled code (Attk.). It is up to a remote verifier and individual enclaves to build a trust hierarchy among "good" enclaves. To do this, trust is extended via SGX remote and local attestation, respectively. The entrusting party does not only verify validity of an SGX attestation report but also check for a correct `MRENCLAVE` value, which uniquely identifies an enclave codebase.

In a typical scenario, a remote verifier wants to establish a trusted path to a user. It therefore extends trust to a specific user app (UA) under its control, which in turn entrusts appropriate secure I/O drivers (Drv). Drivers extend trust to the TB enclave, which does hypervisor attestation as previously outlined. If hypervisor attestation succeeds, trust is implicitly extended to the TB host and all driver host processes together with all trusted paths to user devices. If at any point in the trust hierarchy attestation fails, the affected entities will terminate trusted path attestation.

6.5 User Verification

SGXIO allows a user to locally assess if he is talking with the correct user app via a trusted path. This does not require additional hardware such as an external handheld verification device [42] or similar. Instead, we stick to sharing a secret piece of information between user and user app, similar to [38].

For the sake of simplicity we discuss the common scenario of a trusted screen path and a trusted keyboard path. When the user starts the user app, the user app requests a trusted input path to the keyboard and a trusted output path to the screen from the corresponding drivers. If for any reason one or both trusted path setups fail, the user app terminates with an error. In the case of success, the user app displays the pre-shared secret information via the screen driver to the user. The user verifies this information to get assured of a valid trusted path setup for this user app. Since an attacker does not know the secret information, he cannot fake this notification. This approach requires provisioning secret information to a user app, which seals it for later usage. Provisioning could be done once at installation time in a safe environment, *e.g.,* with assistance of the hypervisor, or at any time via SGX's remote attestation feature.

7. TWEAKING DEBUG ENCLAVES

SGXIO makes heavy use of SGX enclaves. However, Intel's licensing scheme for production enclaves might be too costly for small business or even incompatible with the open-source idea. We show how to level up debug enclaves to behave like production enclaves in our threat model. This requires special handling of SGX remote attestation and sealing.

Recall that the only difference between debug and production enclaves is the presence of SGX debug instructions, which we aim to disable manually. The debug tweak leverages SGX's support for VT-x instruction interception. Via the so-called ENCLS-exiting bitmap the hypervisor can selectively intercept all SGX debug instructions which are ever executed from within a VM. By doing so, the only code which is able to debug enclaves is the trusted hypervisor itself. Hence, we have effectively turned all debug enclaves inside the VM into production equivalents.

Tweaked Cloud Enclaves. This tweak only applies to a setting similar to SGXIO, where a trusted hypervisor is present. In general, this is not the case for cloud scenarios where the cloud provider is untrusted and expected to subvert the hypervisor. In such cases, one has to opt for real production enclaves. Nevertheless, honest server administrators could use the tweak to obtain SGX protection without licensing. This would help in strongly isolating server code and reducing the TCB from the whole system down to the hypervisor and the enclave code.

Remote Attestation. With tweaked debug enclaves, remote attestation requires special care since a remote verifier cannot easily determine whether the debug tweak is correctly enabled or not. For example, an attacker could compromise the hypervisor and manipulate (debug) the TB enclave to issue wrong approvals. Next, the attacker could stealthily debug all enclaves on the system.

To do remote attestation with tweaked debug enclaves, one can run only the TB enclave in production mode and do remote attestation towards it. Once a remote party verified the TB enclave, it can be sure that the hypervisor correctly enforces the tweak for all debug enclaves in the system.

Sealing. Both, non-tweaked and tweaked debug enclaves share the same sealing keys. This is no problem unless an attacker manages to compromise the hypervisor and disables the tweak. Although hypervisor attestation would fail in that case, the attacker would be able to extract all sealing keys by simply debugging all enclaves. To prevent this, one can delegate sealing key derivation to the TB enclave. The TB enclave, running in production mode, only derives actual sealing keys if hypervisor attestation succeeds.

8. CONCLUSION

We present SGXIO, the first SGX-based architecture to support generic trusted paths. Therefore, we augment SGX with a small and trusted hypervisor for setting up a generic trusted path, while SGX helps in protecting user apps from an untrusted OS. We solve the challenge of combining the security domains of SGX and the hypervisor. We do so by attesting the hypervisor with assistance of a TPM towards a special trusted boot enclave, which is bound to the local computer. With SGXIO, both a remote party and a local user can verify security of trusted paths.

Furthermore, we show how SGXIO can omit enclave licensing by making debug enclaves behave like production enclaves. To achieve this, the trusted hypervisor disables SGX debugging instructions for the whole untrusted VM.

SGXIO demonstrates that SGX is not limited to cloud computing and DRM scenarios. SGXIO addresses user-centric application security, making generic trusted paths available to SGX enclaves. This can greatly improve application security, protecting against kernel-level keyloggers and screenloggers, for example. SGXIO is compatible to unmodified legacy OSes and runs on off-the-shelf notebooks.

Acknowledgments

This work was partially supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments".

9. REFERENCES

[1] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for CPU based attestation and sealing. In *HASP'13*, volume 13, Aug. 2013.

[2] I. Anati, F. McKeen, S. Gueron, H. Huang, S. Johnson, R. Leslie-Hurd, H. Patil, C. V. Rozas, and H. Shafi. Intel Software Guard Extensions (Intel SGX), 2015. Tutorial Slides presented at ICSA 2015.

[3] ARM. TrustZone. http://www.arm.com/products/processors/technologies/trustzone/index.php. (accessed 2016-04-04).

[4] J. Beekman. Intel has full control over SGX. https://jbeekman.nl/blog/2015/10/intel-has-full-control-over-sgx/, Oct. 2015. (accessed 2016-03-03).

[5] R. Boivie and P. Williams. SecureBlue++: CPU Support for Secure Executables. Research report, IBM, Apr. 2013. Reference no. RC25369.

[6] D. Champagne and R. B. Lee. Scalable architectural support for trusted software. In *HPCA'16*, pages 1–12, Jan. 2010.

[7] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. In *ASPLOS XIII*, pages 2–13. ACM, 2008.

[8] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic. SecureME: A Hardware-software Approach to Full System Security. In *ICS '11*, pages 108–119. ACM, 2011.

[9] V. Costan, I. A. Lebedev, and S. Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security'16*, pages 857–874, Aug. 2016.

[10] 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 *MICRO'14*, pages 190–202, Dec. 2014.

[11] E. Fernandes, Q. A. Chen, G. Essl, J. A. Halderman, Z. M. Mao, and A. Prakash. TIVOs: Trusted Visual I/O Paths for Android. *University of Michigan CSE Technical Report CSE-TR-586-14*, 2014.

[12] A. Filyanov, J. M. McCune, A. R. Sadeghi, and M. Winandy. Uni-directional trusted path: Transaction confirmation on just one device. In *DSN'11*, pages 1–12, June 2011.

[13] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using Innovative Instructions to Create Trustworthy Software Solutions. In *HASP '13*. ACM, 2013.

[14] Intel Software Guard Extensions Programming Reference, Oct. 2014. Reference no. 329298-002US.

[15] Intel 64 and IA-32 Architectures Software Developer's Manual, Sept. 2015. Reference no. 325462-056US.

[16] Intel Trusted Execution Technology (Intel TXT), Software Development Guide, July 2015. Reference no. 315168-012.

[17] Intel Software Guard Extensions Developer Guide, 2016.

[18] Intel Software Guard Extensions Evaluation SDK for Windows OS. User's Guide, Jan. 2016. Revision 1.1.1.

[19] S. Johnson, D. Zimmerman, and B. Derek. Intel SGX: Debug, Production, Pre-release what's the difference? https://software.intel.com/en-us/blogs/2016/01/07/intel-sgx-debug-production-prelease-whats-the-difference, Jan. 2016. (accessed 2016-04-04).

[20] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *SOSP '09*, pages 207–220. ACM, 2009.

[21] N. Knupffer. Intel Insider – What Is It? (Is it DRM? And yes it delivers top quality movies to your PC). https://blogs.intel.com/technology/2011/01/intel_insider_-_what_is_it_no/, Jan. 2011. (accessed 2016-04-04).

[22] M. Lange and S. Liebergeld. Crossover: Secure and Usable User Interface for Mobile Devices with Multiple Isolated OS Personalities. In *ACSAC '13*, pages 249–257. ACM, 2013.

[23] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li. Building Trusted Path on Untrusted Device Drivers for Mobile Devices. In *APSys '14*, pages 8:1–8:7. ACM, 2014.

[24] D. Liu, E. Cuervo, V. Pistol, R. Scudellari, and L. P. Cox. ScreenPass: Secure Password Entry on Touchscreen Devices. In *MobiSys '13*, pages 291–304. ACM, 2013.

[25] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for Tcb Minimization. In *Eurosys '08*, pages 315–328. ACM, 2008.

[26] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP'13*, page 10, 2013.

[27] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: From General Purpose to a Proof of Information Flow Enforcement. In *SP'13*, pages 415–429, May 2013.

[28] *seL4 Reference Manual, Version 3.0.0*. NICTA, Mar. 2016. https://wiki.sel4.systems/Documentation (2016/04/04).

[29] E. Owusu, J. Guajardo, J. McCune, J. Newsome, A. Perrig, and A. Vasudevan. OASIS: On Achieving a Sanctuary for Integrity and Secrecy on Untrusted Platforms. In *CCS '13*, pages 13–24. ACM, 2013.

[30] B. Parno. Bootstrapping Trust in a "Trusted" Platform. In *HotSec'08*, 2008.

[31] PCI Security Standards Council. Approved PIN Transaction Security Devices. https://www.pcisecuritystandards.org/assessors_and_solutions/pin_transaction_devices. (accessed 2016-04-04).

[32] J. M. M. A. Perrig and M. K. Reiter. Safe Passage for Passwords and Other Sensitive Data. In *NDSS'09*, 2009.

[33] X. Ruan. *Platform Embedded Security Technology Revealed. Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. ApressOpen, 2014.

[34] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing. In *ICS '03*, pages 160–171. ACM, 2003.

[35] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang. TrustICE: Hardware-Assisted Isolated Computing Environments on Mobile Devices. In *DSN'15*, pages 367–378, June 2015.

[36] TCG. *Trusted Platform Module Library. Part 1: Architecture. Family 2.0*. Oct. 2014. Revision 01.16.

[37] T. Tong and D. Evans. Guardroid: A trusted path for password entry. *Mobile Security Technologies*, 2013.

[38] Verified by Visa. https://www.visaeurope.com/making-payments/verified-by-visa/. (accessed 2016-08-10).

[39] S. Weiser and M. Werner. SGXIO: Generic Trusted I/O Path for Intel SGX. *arXiv:1701.01061*, Jan. 2017.

[40] M. Yu, V. D. Gligor, and Z. Zhou. Trusted Display on Untrusted Commodity Platforms. In *CCS '15*, pages 989–1003. ACM, 2015.

[41] Z. Zhou. *On-Demand Isolated I/O for Security-Sensitive Applications on Commodity Platforms*. PhD thesis, Carnegie Mellon University, 2014.

[42] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building Verifiable Trusted Path on Commodity x86 Computers. In *SP'12*, pages 616–630, May 2012.

[43] Z. Zhou, M. Yu, and V. D. Gligor. Dancing with Giants: Wimpy Kernels for On-Demand Isolated I/O. In *SP'14*, pages 308–323, May 2014.