# HECTOR-V: A Heterogeneous CPU Architecture for a Secure RISC-V Execution Environment

Pascal Nasahl
pascal.nasahl@iaik.tugraz.at
Graz University of Technology
Graz, Austria

Robert Schilling
robert.schilling@iaik.tugraz.at
Graz University of Technology
Graz, Austria

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

Stefan Mangard
stefan.mangard@iaik.tugraz.at
Graz University of Technology
Graz, Austria

## ABSTRACT

To ensure secure and trustworthy execution of applications in potentially insecure environments, vendors frequently embed trusted execution environments (TEE) into their systems. Applications executed in this safe, isolated space are protected from adversaries, including a malicious operating system. TEEs are usually build by integrating protection mechanisms directly into the processor or by using dedicated external secure elements. However, both of these approaches only cover a narrow threat model resulting in limited security guarantees. Enclaves nested into the application processor typically provide weak isolation between the secure and non-secure domain, especially when considering side-channel attacks. Although external secure elements do provide strong isolation, the slow communication interface to the application processor is exposed to adversaries and restricts the use cases. Independently of the used implementation approach, TEEs often lack the possibility to establish secure communication to external peripherals, and most operating systems executed inside TEEs do not provide state-of-the-art defense strategies, making them vulnerable against various attacks.

We argue that TEEs, such as Intel SGX or ARM TrustZone, implemented on the main application processor, are insecure, especially when considering side-channel attacks. In this paper, we demonstrate how a heterogeneous multicore architecture can be utilized to realize a secure TEE design. We directly embed a secure processor into our HECTOR-V architecture to provide strong isolation between the secure and non-secure domain. The tight coupling of the TEE and the application processor enables HECTOR-V to provide mechanisms for establishing secure communication channels between different devices. We further introduce RISC-V Secure Co-Processor (RVSCP), a security-hardened processor tailored for TEEs. To secure applications executed inside the TEE, RVSCP provides hardware enforced control-flow integrity and rigorously restricts I/O accesses to certain execution states. RVSCP reduces the trusted computing base to a minimum by providing operating system services directly in hardware.

## KEYWORDS

trusted execution environment, secure I/O, heterogeneous computer architecture, RISC-V

## 1 INTRODUCTION

With the growing demand for complex IT applications, such as autonomous driving or smart city infrastructures, software complexity increases steadily. The codebase of the Linux kernel, for example, increases by 250k lines of code (LOC) each year, reaching 27.8M LOC in 2020 [12]. This is challenging because one can expect roughly 1 to 25 bugs in 1,000 lines of code [41]. While not all of these bugs might be exploitable by an attacker, a growing codebase complexity clearly leads to a larger attack surface.

One strategy to deal with the growing complexity and meet security goals is to isolate all security critical applications using a trusted execution environment (TEE) [23]. A TEE establishes a secure execution environment by creating a safe, isolated space using hardware and software primitives [24]. Most TEE threat models consider a powerful adversary controlling user space applications, the operating system, or even the hypervisor, trying to influence the execution of applications in the trusted environment. To achieve the protection needed for this threat model, TEEs require strong isolation between the TEE and the so-called rich execution environment (REE).

The powerful concept of shifting security critical applications into a trusted execution environment is already adapted by major vendors like Intel, ARM, and Apple. One of the most common TEE design approaches is to create a virtual secure processor in the main application processor by using hardware extensions. Intel SGX [20] and ARM TrustZone [7] take this approach. Contradictory to embedding the TEE tightly into the CPU, Google's Titan [34] and Apple's T2 [3] implements a secure element by externally mounting a dedicated security processor next to the main CPU. However, both TEE design approaches yield different weaknesses. Several recent attacks [13, 14, 17, 28, 29, 37, 49, 54, 66] showed that the isolation of TrustZone and SGX can be bypassed by mounting cache or transient-based side-channel attacks. While dedicated secure elements provide strong isolation between REE and TEE, Google's Titan, e.g., uses a slow SPI connection for communication between the two domains limiting potential use cases. Furthermore, this off-chip communication fabric between REE and TEE is physically exposed to an attacker, making it vulnerable against probing attacks.

Independently of the used TEE design approach, typical TEE implementations offer several weaknesses. First, although a security breach within the TEE is fatal, operating systems deployed in the

secure environment surprisingly do not offer state-of-the-art protection mechanisms like ASLR, guard pages, or stack cookies [15]. Second, most TEEs do not provide architectural features to establish secure communication with I/O devices. The lack of trusted I/O paths in TEE systems is critical because secrets shared between user and TEE are unprotected.

With the Intel Lakefield architecture [35] and an AMD patent [42], major vendors recently announced to introduce heterogeneous multi-core architectures in upcoming processor designs. While the approach of tightly coupling different processor cores on one chip to balance power and energy efficiency is already used by ARM's big.LITTLE technology [6] in mobile platforms, Intel and AMD are planning to introduce this concept in forthcoming computer architectures. This design strategy raises the following research question: *RQ1: "Can the tight coupling of distinct processor cores on a SoC be used to increase the security of trusted execution environments?"*

## Contribution

In this paper, we introduce HECTOR-V, a design methodology utilizing a heterogeneous multi-core architecture to develop secure trusted execution environments. HECTOR-V achieves strong isolation between the secure and non-secure domain and provides protection for various side-channel attacks, such as cache and transient-based attacks, by using a distinct processor of the heterogeneous core cluster for the secure environment. The tight coupling of TEE and REE in the shared SoC infrastructure yields several advantages. In the HECTOR-V architecture, the application processor and TEE are connected through a communication fabric, enabling a high-speed link between the two processors. Additionally, since the TEE is embedded into the SoC, all peripherals integrated into the system are also available for the secure environment. To manage peripheral sharing and protect peripherals from unauthorized accesses, HECTOR-V further introduces a secure I/O path concept. The identifier-based strategy deeply nested into the communication fabric and the peripherals allows a fine granular protection of the attached peripherals. In HECTOR-V, the access permission management is implemented using a hardware-based security monitor. While the security monitor owner can configure a set of access permissions, the other parties in the system can request access to certain peripherals by consulting the security monitor. We further extend this mechanism to dynamically grant and deny access to certain peripherals by introducing the concept of security monitor ownership transfer. To enable various use cases, the owner of the security monitor dynamically can transfer the ownership to other parties.

Furthermore, we introduce RVSCP, a security-hardened RISC-V processor tailored for being used as a TEE. Although the HECTOR-V architecture is generic and independent of the used TEE processor, we show that RVSCP further increases the TEE security by utilizing features of HECTOR-V. RVSCP isolates applications within the trusted execution environment by enforcing the integrity of the control-flow. We combine the control-flow information with the secure I/O mechanism to only grant access to certain peripherals when reaching a predefined control-flow state. To reduce the trusted computing base and therefore the attack surface to a minimum, RVSCP implements operating system features, such as multitasking, directly in hardware.

We demonstrate the HECTOR-V concept by introducing a heterogeneous multi-core architecture for RISC-V. We embedded a RISC-V processor with a control-flow integrity protection unit into the HECTOR-V architecture. To verify the functionality of HECTOR-V and RVSCP design approaches, we use secure boot as a prototype application on our FPGA implementation. In summary, our contributions are:

**HECTOR-V,** a design methodology for developing secure TEEs. HECTOR-V utilizes a heterogeneous multi-core architecture to realize a secure trusted execution environment. The architecture includes a configurable security monitor managing access permissions to specific peripherals. This mechanism allows the system to establish secure communication channels between peripherals, users, and the processor cores. The dynamic transfer of security monitor ownership enables the HECTOR-V architecture to realize several use cases, such as secure boot or executing trusted applications.

**RVSCP,** a security-hardened processor integrated into the HECTOR-V architecture. RVSCP protects applications within the TEE by using a control-flow integrity protection unit and combines this mechanism with the peripheral access protection offered by HECTOR-V. The secure processor provides operating system features in hardware to minimize the trusted computing base.

# 2 BACKGROUND

To protect the processing of sensitive data, such as biometric data or cryptographic keys in an untrusted environment, trusted execution environments (TEE) are used. These secure environments, mostly used in mobile platforms and cloud computing, comprise a combination of hardware and software features, the trusted computing base (TCB), which needs to be trusted by the system [23]. A TEE guarantees the secure execution of trusted applications, the trustlets, even when considering a malicious operating system running in the rich execution environment (REE). This feature can either be realized with a virtual processor approach, using an external secure processor, or embedding a secure processor into the SoC.

## 2.1 ARM TrustZone

ARM TrustZone [7] is a TEE mostly used in mobile platforms. TrustZone protects trustlets executed inside the TEE from malicious accesses and tampering approaches, even from the operating system [45]. Due to the massive deployment of ARM cores in Android phones, TrustZone is commonly used by vendors, such as Samsung with KNOX [46], to provide secure key storage and payment. The strong isolation of TrustZone is achieved by partitioning the system into a secure and non-secure world using hardware extensions. TrustZone enforces this separation by introducing a non-secure (NS) bit directly into the bus architecture. This bit indicates, in which domain the cache, the TrustZone Address Space Controller (TZASC), and other peripherals are working and protects secure world devices from non-secure world accesses [44]. Secure communication between the two domains and the handling of context switches, interrupts, and exceptions are managed by the security monitor. Since the rigorous segregation between the two

security domains is enforced for all SoC components, one physical TrustZone core operates like two virtual cores [63]. To deploy multiple trusted applications in the trusted execution environment, usually, an operating system is mounted in the secure environment. Trusty [27], which is used in Android devices supporting TrustZone, consists of a small kernel and provides a communication interface to the non-secure world. Trusted applications, such as key storage and fingerprint processing, are directly integrated into the kernel by the device vendor. Deploying third-party applications is currently not supported by Trusty. Other small footprint operating systems deployed in TrustZone assisted TEEs are the open-source OP-TEE [25] OS developed by Linaro or Qualcomm's QSEE.

**Security of TrustZone.** The main idea of a TEE is to reduce the trusted computing base to a minimum. Kernels deployed in ARM TrustZone comply with this requirement by only providing a small set of operating system services. Although applications executed in the TEE require strong isolation between the trustlets, operating systems like Trusty and OP-TEE fail to provide state-of-the-art protection mechanisms like ASLR or stack cookies [15]. Due to the insufficient protection of applications executed in the TEE, several attacks, like the Qualcomm QSEE attack [10] were discovered in the past. In this particular attack, an adversary could gain remote code execution by exploiting a single buffer overflow vulnerability in a trustlet. These attacks are critical because they threaten not only the security of the TEE but also the security of the REE. Since Qualcomm's QEE allows the TEE to modify arbitrary memory, an attacker easily could obtain control over the Android system executed in the REE [11]. However, not only the lack of conventional security concepts in TEE kernels is problematic, but TrustZone also is vulnerable against various side-channel attacks. CLKSCREW [51] utilizes the DVFS energy management technique to perform a fault attack retrieving secret keys from the TEE or deploying malicious code. Although TrustZone separates the cache between the secure and non-secure world, past research demonstrated several cache attacks [29, 37, 66].

## 2.2 Intel SGX

Intel SGX [20] provides hardware-based isolation for enclaves by introducing a set of dedicated instructions embedded into the microarchitecture of the CPU. The threat model covers potentially malicious user applications, the operating system, and the hypervisor and protects the enclave content from unauthorized accesses [21]. To verify the program loaded into the enclave by untrusted parties, Intel SGX offers a software attestation service. SGX preserves confidentiality and integrity of external memory by using transparent memory encryption [30].

**Security of SGX.** Although SGX provides protection against physical attackers targeting the external memory, Intel explicitly excludes side-channel attacks in their threat model and recommends developers to prevent such attacks at enclave level [32]. Past research showed that SGX is vulnerable against cache attacks [13, 28, 49], transient attacks [14, 17, 54], and fault attacks such as Plundervolt [43]. In addition to side-channel attacks, trustlets in Intel SGX are also vulnerable against classical software attacks, such as memory vulnerabilities [19, 36, 57]. Furthermore, SGX does not natively provide secure I/O interaction; it requires additional techniques, such as SGXIO [58], to establish secure communication with peripherals.

## 2.3 Google Titan

Google Titan [34] is a secure element providing a hardware-based root of trust for cloud computing. To mitigate attacks against the integrity of the basic firmware or the operating system, Google uses Titan to verify the boot process on server platforms [26]. Titan internally stores signatures of the boot files and uses cryptographic accelerators to verify the loaded boot images. In addition, Titan also provides several cryptographic services to the main CPU. The dedicated chip is mounted next to the main application processor and establishes communication over a slow SPI interconnect. Google currently announced to use the open-source version OpenTitan [39] in the next Titan generations. Similarly, OpenTitan consists of a small RISC-V processor and several cryptographic hardware accelerators. A smaller version of the Titan, the Titan-M, is used by Google in current Pixel phones.

## 2.4 Apple Secure Enclave Processor

Apple's Secure Enclave Processor (SEP) [31], which was introduced with the iPhone 5s, is a dedicated ARM secure coprocessor directly embedded into the main SoC. Similar to ARM TrustZone in mobile platforms, SEP is mainly used to store keys and enable secure processing of biometric data [4]. SEP, which runs the L4-based SEPOS, fully controls several dedicated secure peripherals, such as cryptographic accelerators, and shares other peripherals on the SoC with the application processor. The memory controller allows the secure coprocessor to define isolated memory regions in the main memory. To enable communication between the main application processor and SEP, a mailbox system is used. Since the secure enclave processor also contains a key and signature storage, SEP is also used to verify the boot process of the application processor.

**Security of SEP.** Although SEP and SEPOS implement basic protection mechanisms like stack cookies, guard pages, and memory encryption, state-of-the-art defense strategies such as ASLR are still missing [40]. In contrast to ARM TrustZone, peripherals are either exclusively owned by SEP or are shared, like the power manager or the PLL clock generator. This makes SEP possibly vulnerable against attacks similar to CLKSCREW [51]. Additionally, the decryption keys for some SEP implementations were already leaked [33].

# 3 THREAT MODEL

Our threat model considers the common attack scenarios on TEEs defined by Cerdeira et al. [15]. We are considering an attacker directly exploiting architectural weaknesses of the TEE and the TEE kernel. Here an adversary uses a combination of bugs in the kernel, flaws in the hardware protection mechanism, and missing state-of-the-art defense strategies, such as ASLR or guard pages, to compromise the system. Furthermore, we expect bugs in trustlets, which can be exploited by an attacker over the communication interface between REE and TEE. The security of applications in REE or TEE also can be threatened by using cache-based or transient-based side-channel attacks. Additionally, we are considering a malicious trustlets explicitly trying to attack the SoC. We are extending the

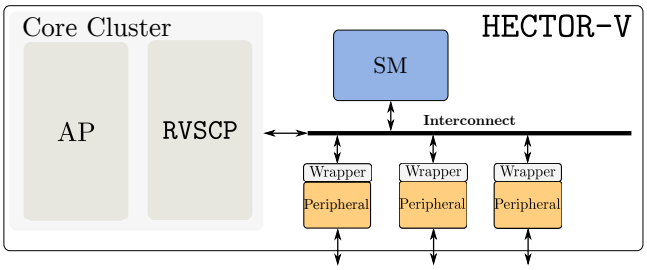


**Figure 1: Overall HECTOR-V design.**

threat model of common TEEs also to cover attacks on peripherals, *i.e.*, illegitimate accesses to secure storage elements, sensitive peripherals such as a fingerprint reader or SPI [1], or protected memory regions. In summary, we expect a powerful attacker having full control over user applications, the operating system, trustlets, or even the hypervisor executed on the application processor.

## 4 DESIGN

This section presents our secure TEE design approach consisting of the HECTOR-V architecture and the secure processor RVSCP. We first introduce HECTOR-V consisting of several architectural features, such as trusted I/O paths, a security monitor, and a secure TEE integration. Then, we introduce RVSCP, a concrete proposal for a secure processor used as a TEE and demonstrate, how RVSCP and HECTOR-V combined, form a secure TEE system. In Section 5, we then demonstrate a prototype of HECTOR-V and RVSCP integrated into our RISC-V base platform, the lowRISC chip.

### 4.1 HECTOR-V Design

The HECTOR-V design proposes architectural features for creating secure TEEs. As seen in Figure 1, the TEE in HECTOR-V is realized by mounting a secure processor directly into the main SoC. This approach is similar to ARM's big.LITTLE technology [6], where independent cores are embedded into the chip. However, instead of using the additional cores for power efficiency, HECTOR-V uses the dedicated processor for security purposes. Although placing a secure processor directly into the SoC increases the area overhead, we argue that this approach is feasible. First, we contend that offering strong isolation between secure and non-secure domain only is possible with a dedicated secure processor for the TEE. This design choice completely separates the instruction pipelines and caches of the cores, resulting in an independent execution flow of REE and TEE applications. Second, services deployed on TEE processors, such as applications handling banking information or processing passwords, are typically rather small. This constrained processing requirement allow us to deploy a tiny secure processor. Therefore, when comparing the TEE processor with a state-of-the-art multi-core processor, the area overhead is negligible.

While integrating a dedicated secure processor into the SoC infrastructure is straightforward, a secure and viable TEE system requires to establish communication channels between REE and TEE, as well as for peripherals enabling user I/O. In the remaining part of this section, we introduce our trusted I/O path design utilizing a hardware-based security monitor and elaborate how these concepts allow REE and TEE to share devices on the SoC and realize features, such as secure storage elements.

*4.1.1 Trusted I/O Paths.* The trusted I/O path mechanism is the central element of the HECTOR-V architecture. This concept allows HECTOR-V to securely share devices on the SoC, establish secure communication channels between external users and the TEE or REE, and implement concepts, like secure storage elements. HECTOR-V establishes trusted I/O paths by assigning an unique identifier to each party of the system, *i.e.*, the processor cores. When accessing a device, like the SD-card or the block RAM (BRAM), the peripheral uses an internal mechanism to verify the identifier. This strategy allows the architecture to enforce access permissions, such as the fingerprint reader only can be accessed by the TEE.

**Identifier.** In HECTOR-V, an identifier is used to distinguish between legitimate and illegitimate accesses to a peripheral. While this concept is similar to ARM TrustZone [7], TrustZone only uses a 1-bit identifier to distinguish between accesses from the secure or non-secure world. The identifier used by our system consists of a core identifier, a process identifier, and a peripheral identifier. While the core identifier permanently and unchangeable is assigned to each processing core, e.g., an identifier for the application processor and one identifier for the secure processor, at design time, the process and peripheral identifier can be assigned by each entity itself.

**Interconnect.** To efficiently transport the identifier from the participants to the peripherals, we integrate the identifier directly into the system interconnect. The AMBA AXI4 [5] protocol, which is used as interconnect by our lowRISC base platform and many other SoC designs, allows to embed up to 1024-bits wide user-defined signals into the protocol. By embedding the identifier into the user-defined signals, which are not used by most AXI devices, the identifier is transported without any protocol overhead. Additionally, this approach assures that the identifier is sent along with the address and data on each AXI request. Since the core identifier is hardcoded directly into the the interconnect interface of the participants, an attacker can not change this identifier from software.

**Peripherals.** HECTOR-V creates a trusted channel between an entity, e.g., the application or the secure processor, and a peripheral by enforcing the identifier-based access check directly in hardware at the peripheral driver. The peripherals, e.g., the hardware implementation of the SD-card controller or the BRAM, use the identifier to filter illegitimate accesses. This scheme, which requires the peripherals to be identifier-aware, is implemented by introducing a lightweight wrapper module for each peripheral instance. In our architecture, the peripheral wrapper module consists of two communication channels: the data channel and the configuration channel. The data channel interface of the peripheral wrapper is directly connected to the AXI4 communication fabric, allowing other parties, e.g., the secure processor, to access the peripheral. By using the configuration channel, the configuration party can set or unset the identifier in the ID field. In HECTOR-V, the data and configuration channels are physically separated by introducing a dedicated, lightweight AXI4-lite [5] communication fabric.

When the identifier of the AXI4 communication request transported over the communication fabric matches the identifier stored in the peripheral wrapper ID field, the access is permitted and the request is directly forwarded to the actual peripheral driver. On an ID mismatch, the firewall mechanism blocks the request and returns an exception over the AXI4 channel, which can be processed by the issuer. If the process or peripheral identifier is set to zero, the peripheral wrapper ignores these ID fields.

The peripheral wrapper resides in two states: claimed or unclaimed. While in the unclaimed state all data channel requests are blocked, in the claimed state only the requests with the matching identifier are permitted. Furthermore, we differentiate between configurable and non-configurable peripheral wrappers. To realize functionalities, like secure storage elements, which must not be accessible by any party except one, non-configurable peripheral wrappers store a hardcoded identifier, consisting of the core and process ID, in the ID field.

*4.1.2 Security Monitor.* The architectural features of identifier, identifier transportation, and peripheral firewalls are the foundation for establishing trusted I/O paths on the SoC. To enforce security policies utilizing the trusted paths, we embed a tiny security monitor into the SoC infrastructure. This hardware-based security monitor (SM) module is responsible for managing access permissions to peripherals. Internally, the SM consists of a table tracking these access permissions. For each peripheral included in the system, the security monitor maintains a table entry tracking the state (claimed or unclaimed) and a list of identifiers allowed to claim the device.

**Security Monitor Owner.** The HECTOR-V architecture introduces the security monitor owner privilege, which is assigned to a certain party at design time. This party solely is responsible for configuring the security monitor. More concretely, only the SM owner is allowed to define, which peripheral can be claimed by which participant. The security monitor stores the identifier of the current owner and accepts configuration commands only from this party.

**Peripheral Claiming and Releasing.** To access a certain peripheral, the requester first needs to claim the peripheral by sending a claim request to the security monitor. Then, the SM checks, if the peripheral currently is unclaimed and verifies that the ID of the issuer is in the list of identifiers permitted to access the peripheral. Finally, the security monitor sends the ID of the issuer to the peripheral wrapper over the AXI4-lite bus and the peripheral firewall sets this identifier in its ID field. Now, the identifier transported in the user-defined signals of each AXI4 request issued by the requester matches the identifier in the ID field and the requester exclusively can access the peripheral. However, if the requester is not allowed to claim the peripheral, *i.e.*, the identifier of the requester is not in the list of privileged entities, the ID verification fails and the security monitor sends an access denied message back to the issuer. If the requester ID is in the table entry of the peripheral, but the peripheral currently is claimed, the security monitor notifies the issuer.

HECTOR-V is a cooperative scheme, meaning, the entity currently claiming a peripheral needs to release it after using it. To release a peripheral, the entity sends a dedicated release command to the security monitor. The security monitor processes this request by clearing the ID field of the peripheral wrapper.

**Peripheral Access Withdraw.** Claiming a peripheral and binding access exclusively to an entity is a powerful concept and establishes a trusted, secure channel between this entity and a peripheral, but it also can be abused. A participant, e.g., the application processor or the secure processor, in control of an attacker, could permanently occupy one or more peripherals resulting in a denial-of-service (DOS) attack. To mitigate such attacks and manage unresponsive participants, the security monitor has the capability to withdraw access from certain peripherals. A simple approach to implement this functionality would be to clear the ID field in the peripheral wrapper. However, this approach is dangerous because it could enable time-of-check to time-of-use (TOCTOU) attacks. For example, when withdrawing access to a peripheral currently processing a secret, the SM owner would be able to retrieve the result of the request.

To securely withdraw access to certain peripherals, HECTOR-V introduces a withdrawing mechanism, which can be triggered by any entity in the system. While a withdraw request issued by the SM owner is always granted, a request from other parties first needs to be approved by the security monitor. When a withdraw request is retrieved by the security monitor, the SM simultaneously starts a timer with a predefined timeout and notifies the owner of the peripheral. The notification of the peripheral owner is realized by introducing a dedicated interrupt line and a interrupt service routine (ISR) provided by the processors for each peripheral. The ISR implements a cleanup function responsible for clearing secrets, stopping current transactions, and gracefully releasing the corresponding peripheral. After the timeout is reached in the security monitor and the peripheral is not released gracefully by the ISR, the SM forcefully releases the peripheral by clearing the ID field in the firewall.

**Security Monitor Ownership Transfer.** A significant advantage of HECTOR-V, compared to other TEE architectures, is the possibility to utilize the TEE infrastructure for various use cases. While in TEE systems, like ARM TrustZone, one entity, e.g., the secure-world, is the exclusive owner of the highest privilege level, HECTOR-V introduces a dynamic ownership transfer mechanism. The security monitor privilege, which allows the SM owner to configure access privileges and release arbitrary peripherals, can be transferred to any other entity by the SM owner. To initiate a SM ownership transfer, the entity sends a request with the identifier of the new owner to the security monitor. The security monitor acknowledges this request by setting the received identifier into the SM owner ID field. To obtain a clear state, we recommend that the security monitor owner first releases all peripherals.

## 4.2 RVSCP Design

To build a secure TEE environment, a processor of the heterogeneous core cluster could be utilized to deploy an operating system. This operating system then is in charge of assigning each trustlet on the core an individual process identifier, annotate data in the cache with the process ID, and manage accesses to peripherals. However, deploying a fairly complex OS, which increases the TCB and the attack surface, is contrary to the usage model of most TEEs. We

argue that trustlets that are commonly deployed in TEEs, such as providing an interface to a cryptographic accelerator, a key storage, or a fingerprint sensor, do not extensively make usage of operating system features. Therefore, we reduce the TCB of RVSCP to a minimum by providing basic operating system services, such as context switches and ID management, directly in hardware. Furthermore, RVSCP utilizes a control-flow integrity (CFI) unit and the secure I/O scheme of HECTOR-V to restrict access to sensitive peripherals.

*4.2.1 Control-Flow Integrity.* To protect a program from attacks targeting to alter the control-flow, CFI schemes are commonly used [52, 55, 65]. These schemes mitigate attacks like ROP [50] or JOP [16] by ensuring that the control-flow of the program can not escape the control-flow graph (CFG) determined at compile time. Enforcing the integrity of the instruction stream can be realized in different degrees of fineness. While some schemes [2, 8, 18] preserve the correctness of the execution flow at basic block level, other techniques [22, 60, 61] maintain the integrity of the control-flow even at instruction granularity.

**Control-Flow Integrity Unit.** RVSCP utilizes the fine-grain Sponge-Based Control-Flow Protection (SCFP) [60] scheme to protect trustlets within the secure processor from attack attempts. The main idea of SCFP is to encrypt a program using a sponge-based authenticated encryption primitive during compile time and decrypt it instruction for instruction at runtime. Decryption of the individual instructions is realized using a dedicated decryption stage in the processor pipeline. To successfully decrypt an instruction, the pipeline stage needs to know the key and an internal state. The SCFP scheme accumulates information over all previously executed instructions in this state. If the integrity of the state is violated, the decryption fails and returns a faulty instruction, which can be detected with a certain probability by the CPU. An attacker modifying instructions, e.g., using fault injection, or inserting additional instructions, alters the state and can be detected by SCFP.

*4.2.2 Hardware Scheduling.* We extend the native SCFP implementation, which allows to execute a bare-metal program on a processor, to support multitasking. One approach to enable multitasking with CFI protected trustlets could be realized purely in software using an operating system. However, since TEE operating systems, such as OP-TEE [25], do not provide state-of-the-art protection mechanisms, such as ASLR or guard pages, mounting an operating system would also increase the attack surface of the TEE. Therefore, similar to Antikernel [67], RVSCP offers hardware features to run multiple trustlets on the processor and reduce the software TCB to a minimum.

**Hardware Scheduling Unit.** RVSCP introduces a hardware entity providing minimal OS functionality for trustlets. This hardware unit is responsible for performing secure context switches between individual trustlets in hardware. The round-robin based scheduling mechanism internally maintains a list of trustlets and after a certain amount of cycles executed, a context switch is conducted. When performing a context switch, the hardware entity stops the current trustlet, stores the architectural state to a secure place, and loads the next architectural state of the next trustlet. Additionally, the hardware context switch mechanism also exchanges the decryption key used for SCFP. Using an individual key for each trustlet yields two major advantages. First, since the programs are encrypted with a different key, only the developer with the correct key can access the plain program, leading to an IP protection mechanism for trustlets. Second, using different keys for trustlets enables strong isolation between the applications. After the context switch, the execution is resumed and the next trustlet is executed.

While this hardware scheduling unit allows the processor to basically consist of several virtual processor cores, only a fixed number of trustlets can run simultaneously on the physical core. However, since most TEEs are anyway limited in their processing power and only a well-chosen set of trustlets is usually deployed in TEEs by the vendor, an upper bound of processes is acceptable. Furthermore, we argue that for simple services typically used in mobile platforms, such as a process handling biometric data for unlocking the device or a process handling the secure key storage, no dedicated operating system is needed. Completely omitting the operating system and providing operating system services using a tiny hardware unit reduces the software TCB to a minimum and would even allow to formally verify the simple hardware unit.

*4.2.3 Control-Flow Integrity with Secure I/O.* The fine granular control-flow integrity unit embedded into the HECTOR-V architecture prevents an attacker from performing control-flow hijack attacks by limiting the control-flow of the program to only valid paths through its control-flow graph. However, while the CFI scheme protects the control-flow by detecting integrity violation of forward- and backward-edges, and thus prevents attacks such as ROP or JOP, data-oriented attacks can not be detected by this countermeasure. In such attacks, an adversary modifies control- or non-control related data to break the security of the system. By manipulating control-related data, such as the condition value in an `if` statement, the attacker indirectly can influence the control-flow of the program. Furthermore, an attacker could leak sensitive data, such as passwords or cryptographic keys, by manipulating addresses in the system. For example, instead of returning the ciphertext over an API to the user, an attacker could modify the address from pointing to the location of the ciphertext in memory to the encryption key stored in a secure storage element instead by exploiting a buffer overflow bug.

To lower the impact of such attacks, RVSCP binds access to certain assets to a certain CFI state. More concretely, only when the CFI protected program reaches a predefined CFI state, the program is permitted to access the distinct peripheral. In RVSCP, this strategy is realized by tunneling each interaction request with a peripheral through a dedicated function with a certain state. RVSCP automatically sets the peripheral identifier of the processor to the current state. Only if this state matches the ID stored in the peripheral wrapper, access to the device is granted. If an attacker calls the peripheral access function outside the valid control-flow graph, the CFI mechanism detects this violation. Additionally, if the adversary crafts an address accessing the asset, the state used as an identifier does not match the identifier of the peripheral and access to the device is restricted.

## 5 IMPLEMENTATION

In this section, we provide a prototype implementation of the HECTOR-V architecture and the secure RVSCP processor. We first

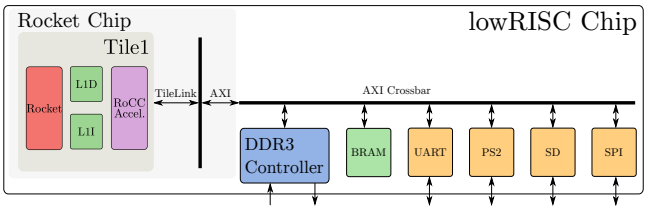


**Figure 2: lowRISC base platform.**

introduce the RISC-V lowRISC chip, which we use as our base platform. Then, we extend this platform with the HECTOR-V features. Finally, we present our RVSCP design and demonstrate how this processor is embedded into the SoC infrastructure.

## 5.1 Base Platform

The prototype implementation is based on the open-source lowRISC [38] project. Internally, the lowRISC chip consists of an instance of the 64-bit RISC-V Rocket chip [9, 56]. The SoC, capable of running Linux, provides an external off-chip DDR3 memory and an on-chip BRAM. As shown in Figure 2, peripherals, such as a SD-card controller and a SPI interface are connected over an AXI4 communication fabric with the processor core.

## 5.2 HECTOR-V

Figure 3 depicts the prototype implementation of the HECTOR-V architecture. The extended lowRISC base platform consists of the Rocket Core application processor ❹ and the secure processor RVSCP integrated into the TEE ❺. By providing an AXI4 master interface ① ②, REE and TEE gain access to various peripherals over the shared communication fabric ③. To differentiate between REE and TEE, the immutable core identifier is directly embedded into this AXI4 master interface. We further introduce a security monitor ❶, a memory protection unit (MPU) ❷, a reset unit ❸, and a secure storage element ❻ as part of the HECTOR-V architecture.

**Interconnect.** The SoC communication infrastructure consists of a shared AXI4 ③ and AXI4-lite Ⓒ interconnect. In our architecture, the AXI4 bus is used to enable interaction between the processing cores and the peripherals. We extended the AXI4 bus protocol and the crossbar to transport the 16-bit user signal along with each bus request. This user signal carries the 1-bit core identifier, the 4-bit process identifier, and the 10-bit peripheral identifier. While communication between peripherals and cores requires a high-speed link, the configuration of the security monitor and the peripheral wrappers only consists of small configuration commands. Hence, we use a lightweight AXI4-lite bus as a configuration channel. The configuration of the security monitor is realized by introducing a point-to-point communication channel between REE and SM Ⓐ and TEE and SM Ⓑ. By using a separate AXI4-lite interconnect Ⓒ, the configuration of the peripheral wrappers can only be initiated by the security monitor. This strategy ensures that neither the TEE nor the REE directly can manipulate the peripheral firewalls; configuration only can be requested over the security monitor module.

**Security Monitor.** To receive commands from REE and TEE, the security monitor implements two AXI4-lite slave interfaces. The protocol used to interact with the security monitor consists of two privileged and four unprivileged commands. With the privileged configuration command, the SM owner configures the table entries of the hardware module. The table includes all peripherals known to the SM, the current claiming status, and a list of identifiers allowed to request access to the device. By using the privileged ownership transfer configuration command, the SM owner can define a new designated SM owner. The permission to issue a privileged configuration command is checked by the security monitor using the SM owner ID field stored in the SM module. The unprivileged commands consist of a claiming and release request command allowing the issuer to gain access to a peripheral. A status command can be used to determine the permission level of the peripheral and if it is currently claimed. To gain access to an already claimed peripheral, the unprivileged withdraw request command can be used. While a withdraw request issued by the SM owner is always granted, the request of the unprivileged participant first needs to be approved. When the withdraw request is granted, the security monitor uses dedicated interrupt lines ❼ to notify the current owner of the peripheral about the withdraw request. For each peripheral, a dedicated interrupt line between SM and TEE or REE is introduced. When the request is approved by the security monitor, the configuration command is forwarded to the peripheral wrapper over the AXI4-lite crossbar Ⓒ.

**Peripheral Wrapper.** An AXI4 read or write request issued by the TEE or REE and transported over the AXI4 interconnect ③ is not directly sent to the peripheral driver. First, a simple logic deployed in the peripheral wrapper checks if the identifier stored in the user signal of the AXI4 request matches the identifier stored in the ID field of the wrapper module. When the process ID or the peripheral ID is set to zero, the access control is only conducted with the core ID, allowing all entities on either the TEE or REE to access the peripheral. Then, if the ID transported in the request matches the ID stored in the module, the request is forwarded to the actual peripheral. However, on an ID mismatch, the peripheral wrapper transports the error code SLVERR to the issuer using the RRESP or BRESP AXI4 signal. In addition to the AXI4 slave interface, the peripheral wrappers also implement an AXI4-lite slave interface. This interface is used by the SM to set or unset the identifier in the ID field of the wrapper.

**Interrupt.** The peripheral wrapper also is responsible for routing the interrupt line of the peripheral to the current peripheral owner. To realize correct interrupt handling, the peripheral wrapper modules consists of one dedicated interrupt line for each participant (REE and TEE). Based on the identifier of the current peripheral owner, the interrupt either is routed to the TEE or the REE.

**Software Support.** To interact with the security monitor, we provide a Linux kernel module for the application processor. This kernel module allows the user processes to claim, release, withdraw, or query the status of a peripheral. Furthermore, the kernel module also provides functionalities to configure the security monitor. To handle interrupts from the security monitor withdraw mechanism, each peripheral driver is extended with a dedicated cleanup interrupt service routine. This ISR is responsible for clearing any secrets, aborting communication channels with other parties, and releasing the peripheral gracefully using the release mechanism.

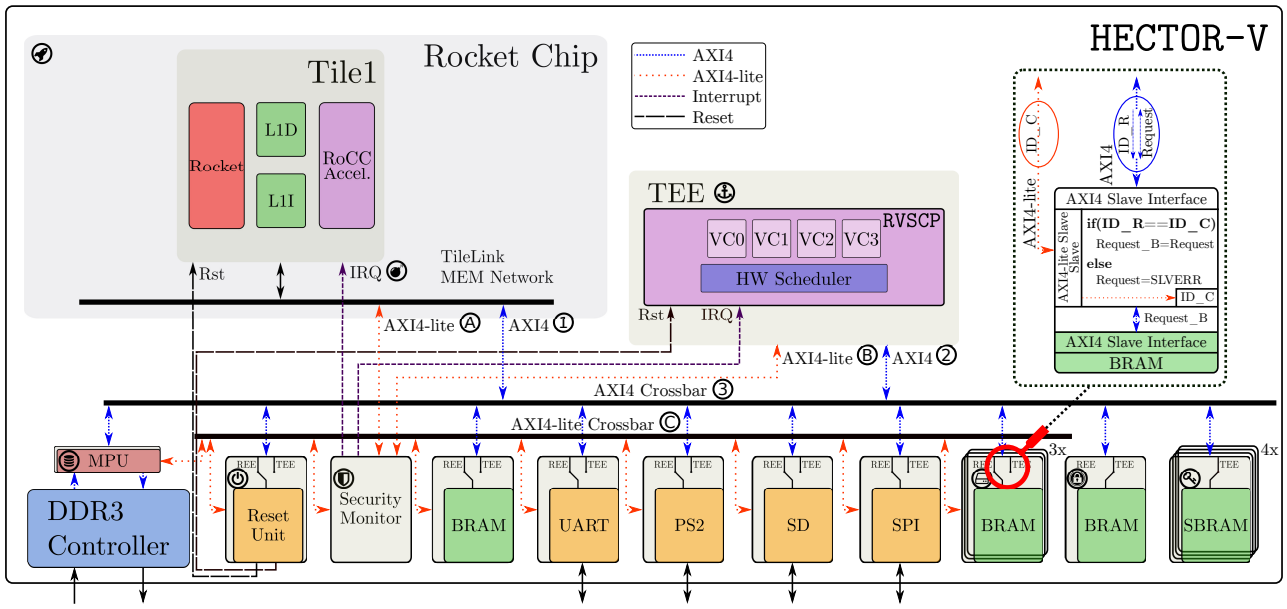


**Figure 3: The HECTOR-V architecture.**

For the TEE, we provide a small library. Similar to the kernel module, this library provides basic functionalities to interact with the security monitor

**Physical Memory Protection.** Isolating a peripheral by binding it exclusively to one entity does not work for the shared, external DDR3 memory. Therefore, the HECTOR-V architecture introduces a memory protection unit (MPU) ⓜ, which is placed directly between the memory controller and the AXI4 bus interface. The MPU can be claimed like any other peripheral by the TEE or REE using the security monitor and the dedicated AXI4-lite slave interface. The party currently claiming the MPU is now able to divide the physical memory into up to 16 regions. These regions can then be either exclusively accessed by one entity or are shared among multiple entities. Each incoming AXI4 read or write access is checked by the memory protection unit using the identifier transported with the request. With this mechanism, a secure storage place for the REE and each virtual core of the RVSCP is enabled and shared communication channels between REE and TEE can be established.

**Reset Unit.** The reset unit ⏻ controls the reset lines of the application and the secure processor. Similar to other peripherals, this entity can be claimed by each participant in the system. The owner of the device can release the reset lines and turn on or off the other entity in the system.

**Secure Storage.** To securely store sensitive data, such as cryptographic keys, biometric data, or user passwords, HECTOR-V introduces a secure storage element 🔑. In contrast to other peripherals, a predefined, immutable identifier consisting of the core ID and the process ID is programmed into the ID field of the wrapper module at design time. Therefore, only the entity with the corresponding identifier can access the storage element. In the prototype, each of the four virtual cores of RVSCP posses an own secure storage.

## 5.3 RVSCP

The RVSCP prototype implementation is based on the 32-bit RISC-V REMUS core [47, 60] already offering the sponge-based CFI unit. The REMUS core originally is based on the Ri5CY core [53], which achieves similar performance as an ARM Cortex-M4 core. We further extend the core with the RVSCP features and embed the processor into the HECTOR-V architecture.

**TEE Infrastructure.** By using the AXI4 master interface ② connected to the AXI4 crossbar ③, the RVSCP is able to interact with the peripherals, such as the UART or PS2 controller. Similar to the main application processor, the secure processor implements an AXI4-lite master interface Ⓑ to configure and transmit peripheral access requests to the security monitor.

**Context Switch.** The hardware scheduler unit is responsible for performing secure context switches. For the RVSCP prototype, the hardware scheduler maintains a list of four slots enabling four virtual cores VC0 ... VC3 on the RVSCP core. On a context switch, the hardware scheduler saves the current SCFP state and the current register file and loads the state, the decryption key, and the register file for the next trustlet. To implement the replacement of the register file, we added four additional register sets to our processor. Note that the register file replacement could also be implemented by storing and loading the content of the registers to memory, e.g., to a secure storage element, to keep the area overhead of the processor low.

To differentiate between the four trustlets executed on the RVSCP, each of the four slots gets assigned an individual process identifier. While the core ID is identical for all slots, the hardware scheduler replaces the process ID directly in the AXI4 and AXI4-lite master interface individually for each slot. By using the same core ID for all four threads, a peripheral could be configured to be accessible by all four trustlets.

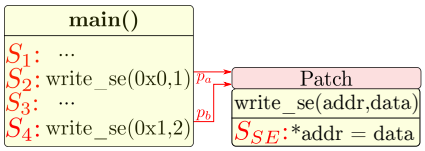


**Figure 4: Access function for secure peripheral accesses.**

**Decryption Keys.** To decrypt the encrypted instruction stream, the SCFP unit needs to know the decryption key for the corresponding trustlet. In the prototype implementation, the key is stored in a dedicated control and status register of each virtual core, which is only accessible from the respective core. To initially load the key into the secure storage, our prototype trustlet consists of a small, unencrypted boot code and the actual, encrypted code. The unencrypted boot code can either generate the key, load the key from the network, or directly from the binary. After storing the key into the key register, the actual encrypted trustlet starts to execute.

**Code Storage.** Each of the virtual processor cores `VC0` ... `VC3` running on the `RVSCP` processor executes code from an on-chip BRAM. While `VC1` ... `VC3` fetch code from a claimable BRAM (🖴), the first virtual processor core fetches its code from a secure code storage element (🔒) exclusively and permanently owned by `VC0`. To utilize one of the virtual cores at `RVSCP` as an enclave, the issuer needs to store the trustlet code on the BRAM of this core. Since the code of the first virtual core is stored in a secure storage element (🔒), this code cannot be changed by the REE or by the other virtual cores of `RVSCP` at any point in time.

**Control-Flow Integrity with Secure I/O.** To implement the concept of binding access to peripherals to a certain CFI state, each AXI4 request leaving the `RVSCP` is annotated with the current CFI state. The processor directly places a compressed form of the current state into the 10-bit peripheral identifier field of the AXI4 user signal. To only allow, e.g., write access to the secure element when reaching a predefined state, the trustlet tunnels all write accesses to this peripheral through the function `write_se`. Since decrypting this function only works, when a certain state $S_{SE}$ is reached, the SCFP scheme automatically patches the state of the callee with the patch value $p_a$ or $p_b$. As seen in Figure 4, the patching mechanism of SCFP ensures that, on each valid call of the access function, state $S_{SE}$ is reached. By using this state as the peripheral identifier, access to a specific peripheral only is granted when reaching state $S_{SE}$ through the access function. Similar to the setup procedure of the decryption keys, the boot code of the trustlet claims the peripheral used in the code by sending the identifier consisting of the core identifier (ID of `RVSCP`), the process identifier (ID of the virtual processor), and the peripheral identifier (compressed CFI state $S_{SE}$) to the security monitor. Now, access to secrets stored in the secure storage element or other sensitive peripherals, like a fingerprint reader, only is permitted when reaching a predefined CFI state. If a trustlet does not need explicit protection of certain peripherals, the peripheral identifier in the peripheral claiming request is set to zero.

## 5.4 Area Overhead

To quantify the area overhead introduced by the additional RISC-V core and the `HECTOR-V` features, we synthesized the lowRISC base platform for a Xilinx Kintex-7 series FPGA. Since the `HECTOR-V` architecture is generic, we synthesized the extended lowRISC platform with the additional `HECTOR-V` features and a selection of different RISC-V cores used as TEE processor. Table 1 shows the total hardware overhead of the `HECTOR-V` architecture consisting of either the `Ri5CY` [53], the `REMUS` [47, 60], or the `FRANKENSTEIN` [48] core used as secure TEE processor. The `REMUS` core, which is based on the `Ri5CY`, implements a decryption unit to offer the Sponge-Based Control-Flow Protection (SCFP) mechanism. `FRANKENSTEIN`, which is an extended version of the `REMUS` core, is a 64-bit RISC-V processor including the SCFP scheme and a pointer protection scheme for secure memory accesses. While the Rocket Core used in the lowRISC platform is capable of running Linux, the processor core still is rather small compared to application processors from vendors like Intel, ARM, and AMD. Therefore, when using a larger processor in the `HECTOR-V` architecture, the relative overhead introduced by the additional secure TEE processor is negligible. In Table 2, we list area number for different components in the `HECTOR-V` architecture using the `RI5CY` core as secure TEE processor.

# 6 USE CASES

This section proposes use cases for the secure TEE design consisting of the `HECTOR-V` architecture and the `RVSCP`. More concretely, we demonstrate how the TEE can be used to boot Linux on the main application processor securely. After the secure boot use case, we show how the `RVSCP` can be utilized to securely execute trustlets. Note that these two scenarios are only a selection of many other use cases that can be realized with `HECTOR-V`.

**Table 1: Lookup table (LUT) overhead of the overall architecture consisting of the `HECTOR-V` features and an additional secure processor compared to the lowRISC base project.**

| Configuration | Area [LUTs] | Area Overhead [%] |
|---|---|---|
| lowRISC base platform | 55,443 | - |
| `HECTOR-V` with `RI5CY` | 63,648 | 14.8 |
| `HECTOR-V` with `REMUS` | 67,024 | 20.89 |
| `HECTOR-V` with `FRANKENSTEIN` | 68,746 | 23.99 |

**Table 2: Number of lookup tables for different features of the `HECTOR-V` architecture.**

| Component | Area [LUTs] | Area [%] |
|---|---|---|
| Rocket Chip | 33,341 | 52.38 |
| `RI5CY` | 5,780 | 9.08 |
| Security Monitor | 446 | 0.7 |
| Peripheral Wrapper | 43 | 0.07 |
| AXI4 Crossbar | 3,052 | 4.79 |
| AXI4-lite Crossbar | 93 | 0.14 |

## 6.1 Secure Boot

On the unmodified lowRISC platform, a zero stage bootloader (ZSBL) permanently stored in the on-chip BRAM first loads the Berkeley boot loader (BBL) from the external SD-card. Then, the ZSBL hands over control to the BBL first stage bootloader (FSBL). The BBL, which is linked against the Linux kernel, fetches the Linux image and sets up the environment by configuring the hardware threads (HARTS) and the memory. Finally, the Linux operating system starts. However, loading the FSBL and the Linux image directly from the SD-card to boot up the device is problematic in many ways. An attacker with physical access to the device can boot arbitrary code by simply modifying the unprotected boot files stored on the SD-card. This attack methodology also can be used in an online attack by overwriting the boot files. To only allow authenticated software to boot, vendors frequently embed a secure boot mechanism into their systems. Here, a chain-of-trust is generated by authenticating each boot file before executing it.

In our use case implementing secure boot using features of the HECTOR-V architecture, the first virtual core VC0 of RVSCP is the designated security monitor owner. At design time, the reset unit of the system is configured to release the reset line of RVSCP and keep the REE processor halted when applying power to the SoC. Additionally, the security monitor owned by VC0 is unconfigured, except for the reset unit which is claimed by the SM owner. When applying power to the device, VC0 starts to execute the ZSBL stored in the secure code storage ⓔ. Since the secure code storage is permanently and exclusively owned by VC0, the ZSBL is isolated from the other parties and only VC0 can update this code using an update mechanism. Due to these strong protection mechanisms, the ZSBL stored in the secure code storage is the system root-of-trust. The ZSBL code first claims the SD-card controller and configures the memory protection unit of the DDR3 memory. Then, the ZSBL fetches the BBL from the SD card and stores it to the external memory. Before passing control to the BBL, the ZSBL determines the hash value of the loaded BBL image. When this hash value does not match the expected hash value stored in the secure storage element ⓒ of VC0, the boot process is aborted. Again, an update of the expected hash value only can be initiated by the first virtual processor core of RVSCP because the secure storage element exclusively is owned by this party and can not be claimed by any other party at any point in time. If the verification of the BBL was successful, the ZSBL releases the SD-card driver, gives the main application processor access to the DDR3 memory region by configuring the MPU, and triggers the reset unit to start the REE. Now, the Rocket Core starts to execute the modified BBL from the external memory. The modified BBL then requests access to the SD-card controller, loads the Linux image from the SD-card, and verifies the loaded image by comparing the computed hash value with the expected hash value. Finally, if the verification process succeeds, the Linux operating system starts and can claim the first peripherals by sending requests to the security monitor. Since each stage of the boot process now is cryptographically verified, the system is in a genuine state. Now, the VC0 passes the SM owner privilege to the main application processor (AP). This change is initiated by sending the identifier of the AP to the SM using the ownership transfer command. Although the REE is now in full control of the system, e.g., is able to switch of the secure processor using the reset unit, secrets stored in the secure storage ⓒ and the secure code storage ⓔ are still protected from AP accesses.

## 6.2 Trustlets

In this use case, we utilize the virtual processor VC1 of RVSCP to execute a trustlet. We compile the trustlet with a LLVM-based toolchain supporting the SCFP scheme and encrypt the program with the key $K$. To interact with the outside and to store secrets, the trustlet uses the embedded UART controller and the secure storage element. The secure storage element is protected from malicious accesses by tunneling all requests through a dedicated access function and binding the access to it to the CFI state $S_{SE}$. We define access to the UART controller as uncritical. Therefore the interaction with this controller is unprotected.

First, the application processor switches off the RVSCP by utilizing the reset unit. Then, the processor claims a claimable BRAM ⓓ and stores the code, consisting of an unencrypted small boot code and the encrypted trustlet, to this storage. After the code is saved, the AP sets VC1 as owner of the code storage in the security monitor. Furthermore, the MPU is configured to provide a memory region owned by the trustlet to use as RAM and a shared memory region to establish a communication channel between REE and the trustlet. Finally, the RVSCP is started by releasing the reset line of the secure processor and code gets executed. To initialize key $K$ and state $S_{SE}$, the boot code of the trustlet writes the decryption key into the key register of $VC1$ and claims the secure key storage element by setting the compressed state $S_{SE}$ into the peripheral identifier field. Now, control is passed to the SCFP protected trustlet code and the decryption stage of the processor decrypts the code using the key. To access the UART controller, the trustlet needs to claim the controller by sending a claim request to the SM. The secure storage element is accessed by using the dedicated access function. By setting the compressed state directly into the peripheral ID field of each AXI4 request leaving the processor, access to the secure storage element only is permitted when reaching the predefined state $S_{SE}$. The communication channel established using the shared, external memory allows the REE and trustlet to exchange information. When the designated SM owner, the AP, withdraws access to the UART controller, the claimable BRAM, or the shared memory region, the trustlet is notified using an interrupt. This interrupt is handled by the ISR implemented on the trustlet. The ISR clears all secrets, aborts communication with the UART controller, gracefully releases the peripheral, and enters a predefined IDLE state.

## 6.3 Security Discussion

For the security analysis of our heterogeneous architecture offering a secure environment, we consider three attacker models.

**Exploitable Trustlet.** The first attack model considers a trustlet containing a bug exploitable by an adversary. Similar to recent attacks [10, 11], the bug, e.g., a buffer overflow, could be exploited over the communication API between REE and TEE. RVSCP protects against this attacker model by offering a CFI unit mitigating against control-flow hijacking attacks on forward- and backward-edges. Furthermore, RVSCP limits the impact of a data-oriented attack by

restricting access to sensitive peripherals by combining the CFI unit with the secure I/O concept.

**Malicious Trustlet** The second attacker model considers a trustlet containing explicit code to attack the architecture directly. To prevent an adversary from performing a DOS attack by permanently claiming all peripherals, HECTOR-V offers a withdraw mechanism. Since the core, process, and peripheral ID are managed by the hardware, a malicious trustlet can not change these IDs in software to gain unauthorized access to peripherals. While the heterogeneous architecture prevents an attacker from performing cache and transient-based attacks between REE and TEE, RVSCP needs additional countermeasures, such as flushing the microarchitectural state [64] on a context switch, to also protect against cache attacks. Due to the simplicity and openness of RVSCP, transient-based attacks can not be performed by an adversary. RVSCP protects trustlets within the processor from each other by using a hardware mechanism to perform context switches, use different encryption keys, and use a separate code storage and RAM region for each trustlet.

**Malicious Security Monitor Owner.** This attacker model considers a malicious SM owner targeting the other entities. The concept of setting the ID immutable into the ID field of critical elements prevents the SM owner from accessing secrets, such as keys, hash values, or the secure boot code. In HECTOR-V, withdrawing access to specific peripherals can only be initiated by using the dedicated withdrawing command. Since the withdrawing mechanism notifies the current owner of the peripheral about the incoming withdraw procedure, sensitive data can be first cleared. To prevent the SM owner to switch off a different entity in the system and then access secrets stored in the peripherals, the reset unit could, similar to the withdrawing mechanism, notify the entity about the incoming reset. Except for resetting participants and withdrawing peripherals, the execution of trustlets can not be interrupted by the SM owner.

## 7 RELATED WORK

Concurrent developed with the HECTOR-V architecture, SiFive recently introduced WorldGuard [62]. In WorldGuard, each core gets assigned a world ID and each process on the core can be annotated with a process ID. Similar to HECTOR-V, the ID is transported using the interconnect and requests from participants are filtered by peripherals, the memory, and the caches. However, both architectures differ in various design choices. First, HECTOR-V uses a hardware-based security monitor which only can be configured by one party. In contrast to WorldGuard, the security monitor ownership can be dynamically transferred to other parties allowing flexible use cases. Additionally, the security monitor allows each participant in HECTOR-V to request access to certain peripherals and request access to already claimed peripherals using a withdraw request. We further propose a concrete secure processor design utilizing features of the heterogeneous architecture to create a trusted execution environment. Moreover, we comprehensively describe the hardware-software interaction and demonstrate features of HECTOR-V by introducing several use case scenarios.

## 8 CONCLUSION AND FUTURE WORK

In this paper, we proposed HECTOR-V, a secure TEE design strategy using a heterogeneous CPU architecture. Our design establishes secure paths between peripherals and the cores by tagging each party with an identifier. The peripherals enforce access permissions by checking the identifier, which is transported along with each bus request. To configure these access permissions, we integrate a hardware-based security monitor into the architecture. The security monitor, which exclusively can set permissions, is owned by a configuration party. By allowing to transfer this ownership to other parties, HECTOR-V allows flexible permission management. In contrast to similar design approaches, we provide a notifier-based mechanism to withdraw access to certain peripherals securely. We further introduce RVSCP, a security-hardened CPU design tailored for our architecture. RVSCP combines a fine-granular control-flow integrity scheme with the secure I/O concept of HECTOR-V to restrict access to assets. To complete our TEE design, we introduce secure data and code storage elements, a reset unit, and a memory protection unit. We examine the features of our architecture in a secure boot and enclave scenario.

**Future Work.** To offer a feature commonly used by TEEs, our platform can be extended with a remote attestation mechanism. This feature could be integrated into RVSCP by utilizing the SCFP scheme [59]. Furthermore, the security monitor can be extended to also support fine-granular access permissions, such as read or write only. Additionally, as already discussed in Section 6.3, RVSCP needs to be extended to mitigate against cache-based attacks by either flushing the cache or integrating ID checks in the cache. Although the identifier-based system guarantees that peripherals only can be accessed by certain parties, communication with external devices attached are not authenticated. The HECTOR-V architecture may be extended with a mechanism to authenticate externally attached devices to ensure that, e.g., the TEE only is communicating with trusted devices.

## ACKNOWLEDGMENTS

Intentionally left blank.

## REFERENCES

[1] 2018. CVE-2016-10423.
[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.* 13 (2009). https://doi.org/10.1145/1609956.1609960
[3] Apple. 2020. *About the Apple T2 Security Chip.* Accessed: 2020-07-10.
[4] Apple. 2020. *Product security certifications, validations, and guidance for SEP: Secure Key Store.* Accessed: 2020-07-11.
[5] ARM. 2019. AMBA AXI and ACE Protocol Specification. *arm.com* (2019).
[6] ARM. 2020. *Processing Architecture for Power Efficiency and Performance.* Accessed: 2020-07-07.
[7] Architecure ARM. 2009. Security technology building a secure system using trustzone technology (white paper). *ARM Limited* (2009).
[8] Divya Arora, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. 2006. Hardware-Assisted Run-Time Monitoring for Secure Program Execution on Embedded Processors. *IEEE Trans. Very Large Scale Integr. Syst.* 14 (2006). https://doi.org/10.1109/TVLSI.2006.887799
[9] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. The Rocket Chip Generator. *EECS Department, University of California, Berkeley, Technical Report* (2016).

[10] Gal Beniamini. 2016. *QSEE privilege escalation vulnerability and exploit (CVE-2015-6639).* Accessed: 2020-05-13.

[11] Gal Beniamini. 2016. *War of the Worlds - Hijacking the Linux Kernel from QSEE.* Accessed: 2020-05-13.

[12] Swapnil Bhartiya. 2020. *Linux in 2020: 27.8 million lines of code in the kernel, 1.3 million in systemd.* Accessed: 2020-07-07.

[13] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Workshop on Offensive Technologies – WOOT*.

[14] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*.

[15] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. 2020. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA*.

[16] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented programming without returns. In *Conference on Computer and Communications Security – CCS*. https://doi.org/10.1145/1866307.1866370

[17] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. 2020. SgxPectre: Stealing Intel Secrets From SGX Enclaves via Speculative Execution. *IEEE Secur. Priv.* 18 (2020). https://doi.org/10.1109/MSEC.2019.2963021

[18] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. 2016. HCFI: Hardware-enforced Control-Flow Integrity. In *Conference on Data and Application Security and Privacy – CODASPY*. https://doi.org/10.1145/2857705.2857722

[19] Tobias Cloosters, Michael Rodler, and Lucas Davi. 2020. TeeRex: Discovery and Exploitation of Memory Corruption Vulnerabilities in {SGX} Enclaves. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*.

[20] Intel Corporation. 2019. Intel 64 and IA-32 Architectures Software Developer's Manual. (2019).

[21] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *ePrint 2016/86* (2016).

[22] Ruan de Clercq, Johannes Götzfried, David Übler, Pieter Maene, and Ingrid Verbauwhede. 2017. SOFIA: Software and control flow integrity architecture. *Comput. Secur.* 68 (2017). https://doi.org/10.1016/j.cose.2017.03.013

[23] Jan-Erik Ekberg, Kari Kostiainen, and N. Asokan. 2014. The Untapped Potential of Trusted Execution Environments on Mobile Devices. *IEEE Secur. Priv.* 12 (2014). https://doi.org/10.1109/MSP.2014.38

[24] Jan-Erik Ekberg, Kari Kostiainen, and N Asokan. 2014. The untapped potential of trusted execution environments on mobile devices. *IEEE Security & Privacy* 12 (2014).

[25] Trusted Firmware. 2020. *OP-TEE.* Accessed: 2020-05-12.

[26] Google. 2020. *Titan in depth: Security in plaintext.* Accessed: 2020-07-07.

[27] Google. 2020. *Trusty TEE.* Accessed: 2020-06-07.

[28] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security, EUROSEC 2017, Belgrade, Serbia, April 23, 2017*. https://doi.org/10.1145/3065913.3065915

[29] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. 2016. Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures. In *IEEE Symposium on Security and Privacy – S&P*. https://doi.org/10.1109/SP.2016.11

[30] Shay Gueron. 2016. A Memory Encryption Engine Suitable for General Purpose Processors. *ePrint 2016/204* (2016).

[31] Apple Inc. 2020. *Security enclave processor for a system on a chip.* Accessed: 2020-06-07.

[32] Intel. 2017. *Intel SGX and Side-Channels.* Accessed: 2020-05-23.

[33] The iPhone Wiki. 2020. *Greensburg 14G60 (iPhone6,1).* Accessed: 2020-07-07.

[34] Scott Johnson, Dominic Rizzo, Parthasarathy Ranganathan, Jon McCune, and Richard Ho. 2018. Titan: enabling a transparent silicon root of trust for Cloud. In *Hot Chips: A Symposium on High Performance Chips*.

[35] Sanjeev Khushu and Wilfred Gomes. 2019. Lakefield: Hybrid cores in 3D Package. In *2019 IEEE Hot Chips 31 Symposium (HCS), Cupertino, CA, USA, August 18-20, 2019*. https://doi.org/10.1109/HOTCHIPS.2019.8875641

[36] Jae-Hyuk Lee, Jin Soo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. 2017. Hacking in Darkness: Return-oriented Programming against Secure Enclaves. In *USENIX Security Symposium*.

[37] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*.

[38] LowRISC. 2019. *lowRISC Chip.* Accessed: 2020-05-07.

[39] lowRISC. 2020. *OpenTitan Open source silicon root of trust (RoT).* Accessed: 2020-06-22.

[40] Tarjei Mandt, Mathew Solnik, and David Wang. 2016. Demystifying the secure enclave processor. *Black Hat Las Vegas* (2016).

[41] Steve McConnell. 2004. *Code complete.* Pearson Education.

[42] Elliot H. MednickEdward McLellan. 2020. Instruction subset implementation for low power operation. *US10698472B2* (2020).

[43] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based fault injection attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP)*.

[44] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. 2016. TrustZone Explained: Architectural Features and Use Cases. In *2nd IEEE International Conference on Collaboration and Internet Computing, CIC 2016, Pittsburgh, PA, USA, November 1-3, 2016*. https://doi.org/10.1109/CIC.2016.065

[45] Sandro Pinto and Nuno Santos. 2019. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Comput. Surv.* 51 (2019). https://doi.org/10.1145/3291047

[46] Samsung. 2020. *KNOX and ARM®TrustZone®.* Accessed: 2020-05-07.

[47] David Schaffenrath. 2016. Fault-Attack Secure Processor Design. *Graz University of Technology* (2016).

[48] Robert Schilling, Mario Werner, Pascal Nasahl, and Stefan Mangard. 2018. Pointing in the Right Direction - Securing Memory Accesses in a Faulty World. In *Annual Computer Security Applications Conference – ACSAC*. https://doi.org/10.1145/3274694.3274728

[49] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Detection of Intrusions and Malware & Vulnerability Assessment – DIMVA*. https://doi.org/10.1007/978-3-319-60876-1_1

[50] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Conference on Computer and Communications Security – CCS*. https://doi.org/10.1145/1315245.1315313

[51] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. 2017. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *USENIX Security Symposium*.

[52] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX Security Symposium*.

[53] Andreas Traber, Florian Zaruba, Sven Stucki, Antonio Pullini, Germain Haugou, Eric Flamand, Frank K Gurkaynak, and Luca Benini. 2016. PULPino: A small single-core RISC-V SoC. In *3rd RISCV Workshop*.

[54] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking transient execution through microarchitectural load value injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*.

[55] Zhi Wang and Xuxian Jiang. 2010. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *IEEE Symposium on Security and Privacy – S&P*. https://doi.org/10.1109/SP.2010.30

[56] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. 2011. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA*. Technical Report. EECS Department, University of California, Berkeley.

[57] Nico Weichbrodt, Anil Kurmus, Peter R. Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *European Symposium on Research in Computer Security – ESORICS*. https://doi.org/10.1007/978-3-319-45744-4_22

[58] Samuel Weiser and Mario Werner. 2017. SGXIO: Generic Trusted I/O Path for Intel SGX. In *Conference on Data and Application Security and Privacy – CODASPY*. https://doi.org/10.1145/3029806.3029822

[59] Mario Werner. 2020. *System Architectures and Techniques for Efficient, Secure, and Trusted Code Execution.* Accessed: 2020-08-13.

[60] Mario Werner, Thomas Unterluggauer, David Schaffenrath, and Stefan Mangard. 2018. Sponge-Based Control-Flow Protection for IoT Devices. In *European Symposium on Security and Privacy – EuroS&P*. https://doi.org/10.1109/EuroSP.2018.00023

[61] Mario Werner, Erich Wenger, and Stefan Mangard. 2015. Protecting the Control Flow of Embedded Processors against Fault Attacks. In *Smart Card Research and Advanced Applications – CARDIS*. https://doi.org/10.1007/978-3-319-31271-2_10

[62] Bob Wheeler. 2019. SIFIVE SECURES RISC-V. *Microprocessor report* (2019).

[63] Johannes Winter. 2008. Trusted computing building blocks for embedded linux-based ARM trustzone platforms. In *Conference on Computer and Communications Security – CCS*. https://doi.org/10.1145/1456455.1456460

[64] Nils Wistoff, Moritz Schneider, Frank K. Gürkaynak, Luca Benini, and Gernot Heiser. 2020. Prevention of Microarchitectural Covert Channels on an Open-Source 64-bit RISC-V Core. *arXiv abs/2005.02193* (2020).

[65] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *USENIX Security Symposium*.

[66] Ning Zhang, He Sun, Kun Sun, Wenjing Lou, and Yiwei Thomas Hou. 2016. CacheKit: Evading Memory Introspection Using Cache Incoherence. In *European Symposium on Security and Privacy – EuroS&P*. https://doi.org/10.1109/EuroSP.2016.34

[67] Andrew D. Zonenberg and Bülent Yener. 2016. Antikernel: A Decentralized Secure Hardware-Software Operating System Architecture. In *Cryptographic Hardware and Embedded Systems – CHES*. https://doi.org/10.1007/978-3-662-53140-2_12