Journal of Hardware and Systems Security (2020) 4:329–342
https://doi.org/10.1007/s41635-020-00108-8

Towards Designing a Secure RISC-V System-on-Chip: ITUS

Vinay B. Y. Kumar[1] · Suman Deb[1] · Naina Gupta[1] · Shivam Bhasin[1] · Jawad Haj-Yahya[2] · Anupam Chattopadhyay[1] · Avi Mendelson[2]

Received: 16 June 2020 / Accepted: 13 October 2020 / Published online: 10 November 2020

Abstract
A rising tide of exploits, in the recent years, following a steady discovery of the many vulnerabilities pervasive in modern computing systems has led to a growing number of studies in designing systems-on-chip (SoCs) with security as a first-class consideration. Following the momentum behind RISC-V-based systems in the public domain, much of this effort targets RISC-V-based SoCs; most ideas, however, are independent of this choice. In this manuscript, we present a consolidation of our early efforts along these lines in designing a secure SoC around RISC-V, named ITUS. In particular, we discuss a set of primitive building blocks of a secure SoC and present some of the implemented security subsystems using these building blocks—such as secure boot, memory protection, PUF-based key management, a countermeasure methodology for RISC-V micro-architectural side-channel leakage, and an integration of the open keystone-enclaves for TEE. The current ITUS SoC prototype, integrating the discussed security subsystems, was built on top of the lowRISC project; however, these are portable to any other SoC code base. The SoC prototype has been evaluated on an FPGA.

Keywords Secure SoC · Design-for-security · Threat modeling · RISC-V · Secure boot · Side-channel attack countermeasures · Memory protection · TEE · PUF

1 Introduction

SoC security is emerging as a complex challenge across all layers of abstraction, starting from security protocol analysis, all the way down to robust circuit and device implementation. Despite proclaimed security guarantees, current commercial SoCs are routinely exposed with critical security flaws and unforeseen information side-channels. Examples of such attacks include the following: Rowhammer [1] (kernel access privilege escalation), Meltdown [2] (disclosure of unauthorized memory regions), and Foreshadow [3] (enclave attestation keys extraction). Such attacks can be orchestrated with limited or no privilege such as through a web browser script.

Researchers are working towards a robust and secure SoC design, while keeping the performance constraints in consideration. The performance overhead, when countermeasures for these attacks were deployed, is non-negligible, workload-, and platform-dependent [4]. Despite these concerted efforts towards mitigation, there are often newly

✉ Vinay B. Y. Kumar
vinayby@iitbombay.org

Suman Deb
sumandeb@ntu.edu.sg

Naina Gupta
naina003@e.ntu.edu.sg

Shivam Bhasin
sbhasin@ntu.edu.sg

Jawad Haj-Yahya
jhajyahya@ethz.ch

Anupam Chattopadhyay
anupam@ntu.edu.sg

Avi Mendelson
mendlson@technion.ac.il

[1] Nanyang Technological University, Singapore, Singapore
[2] Technion - Israel Institute of Technology, Haifa, Israel

discovered attacks on top of these countermeasures. For example, kernel hardening can prevent Meltdown attack, but exposes vulnerability to side-channel attacks [5]. It is also conjectured that speculative execution side-channels in modern CPU micro-architectures might be impossible to patch through purely software techniques [6]. Naturally, these concerns require a thorough revisit of design principles for secure SoC design. Designing with security as a first class constraint [7] along with performance, resource, and other considerations allows for a better management of overheads of security, compared to security implemented as an after-thought. Practices from cryptography to achieve provable security through secure composition [8] are also being explored.

Quite a few works have been done in recent times to put these ideas into practice and develop a secure SoC ground-up, based on a standard Instruction-Set Architecture (ISA). Examples include AEGIS [9], from among the earliest efforts from academia, to the more recent Sanctum [10], Keystone [11], TIMBER-V [12], including security enhancements to existing RISC-V SoCs (e.g., Shakti-T [13]) (more in Section 6).

All of these works are based on the common principles of analyzing the threat models, developing security building blocks, establishing root-of-trust, which can be denoted as a *top-down pass*. The subsequent *bottom-up pass* is to begin with a side-channel-resistant implementation and establish trust between layers of abstraction until the application layer. In the following, we discuss the corresponding terminologies, to be used throughout this manuscript.

Let $\mathbb{H}$ represents the set of all hardware components of an SoC, and $\mathbb{S}$ the set of all software components. Furthermore, let $\mathbb{H}_I$ and $\mathbb{H}_E$ refer to internal (on-chip, including any microcode) and external (off-chip) components; and, let $\mathbb{S}_M$, $\mathbb{S}_S$, and $\mathbb{S}_U$ refer to software components executing in privilege modes—*M*achine, *S*upervisor, and *U*ser, in the order of decreasing privilege—respectively, such that $\mathbb{S} = \mathbb{S}_M \cup \mathbb{S}_S \cup \mathbb{S}_U$. Let $\mathbb{N} \subset \mathbb{H}$ represent the set of all interconnects (and related resources) of an SoC (including IO / debug ports). The trusted computing base (TCB) of an SoC is defined as the set, $\mathbb{TCB} \subset \mathbb{H} \cup \mathbb{S} \cup \mathbb{N}$, whose members are either unconditionally trusted (to various degrees) or in whom trust is ensured by design. It is also understood that the degree of trust (to the extent one can assert the lack of weaknesses) between members in $\mathbb{TCB}$ varies (e.g., for $e_1, e_2 \in \mathbb{TCB}$, size($e_1$)>size($e_2$) usually means trust in (e_2) is easier to assert, and $e \in \mathbb{H}$ is harder to attack compared to $e \in \mathbb{S}$).

A traditional presumption for $\mathbb{TCB}$ has been $\mathbb{H} \cup \mathbb{S}_M$ (i.e., hardware is implicitly trusted and so is the software executing in the *M*-mode)—however, this presumption is not sound and the $\mathbb{TCB}$ must strive to be smaller with additional measures implemented to ensure trust by design for each member of $\mathbb{TCB}$.

Architecting a secure SoC begins with a declaration or identification of the threat model (TM) considering the target deployment scenarios. A list of potential attack vectors (AVs) under this TM is then drawn up including identification of corresponding entry-points and enabling conditions for the attack vectors—the Attack Surface. Integrating countermeasures (CMs) to "cover" this Attack Surface is the next step. Implementations of CMs can themselves introduce additional vulnerabilities (e.g., a side-channel leakage). The attack surface could be expected to expand during the lifetime of the device so there must also be generic mechanisms in place to introduce CMs to deal with certain classes of unknown AVs. Once the attack surface is established, securing an SoC would involve securing (or implementing CMs corresponding to the AVs for) all individual components of the $\mathbb{TCB}$. While having a library of all possible countermeasures for each component of $\mathbb{H}$, $\mathbb{S}$, and $\mathbb{N}$, in the $\mathbb{TCB}$, pertaining to various attacks is a valuable pursuit, routinely integrating such collections of CMs would be an overkill. A measured approach to integrate CMs would also include a clear understanding of "trust delegation" between different abstraction layers, thereby seeking a "minimum cover[age]" in terms of the individual combination of countermeasures that are "necessary and sufficient" to thwart the enumerated AVs. For instance, a CM that plugs the cache-timing side-channel leakage would obviate the need for several other countermeasures that are otherwise implemented elsewhere (e.g., somewhere in ($\mathbb{S}$)). A systematic ontological study—relating weaknesses and weakness chains[1] in the system (the $\mathbb{TCB}$ in particular) and how the CMs cover these weaknesses while also keeping track of security properties—would help in arriving at an optimal CM integration plan.

1.1 Contributions

In this manuscript, we present a consolidation of our work towards developing a RISC-V-based secure SoC prototype, ITUS [14].

1. A Secure Boot Protocol ensuring first-instruction-integrity, supporting both classical [15] and post-quantum public-key [16] signature schemes
2. A Key Management Framework around a PUF [17]
3. Memory Protection Unit to ensure confidentiality and integrity properties [18]

[1] https://cwe.mitre.org/documents/glossary/index.html

4. Integrating Keystone Enclaves [11] with the hardware Root-of-Trust in ITUS
5. A new side-channel attack countermeasure (Section 5) strategy proposal and evaluation of an instance of the proposal with a RISC-V out-of-order core.

1.2 Organization

The manuscript is organized as follows. Section 2 discusses some ways to approach SoC threat modeling in general and identifies the specific model presumed for ITUS. Section 3 lists some of the building blocks and elements of a secure SoC while Section 4 discusses how these blocks are implemented and integrated to form specific security subsystems in ITUS. One of the major security subsystems is the processor micro-architecture, and Section 5 discusses countermeasure strategies at this level, with preliminary independent evaluations using a RISC-V out-of-order core. A listing of related works towards secure SoC design and conclusion follow.

2 SoC Threat Models

An SoC being composed of heterogeneous components has a vast attack surface due to wide range of potential vulnerabilities. The varied nature of vulnerabilities prevents the prospect of a one-solution-fit-all. Also, designing a system which addresses all the vulnerabilities will be close to impossible and impractical due to high overheads. A commonly used approach is to define threat models considering the application scenario and find countermeasures which address those threats. For example, Common Criteria evaluations require a security target against which a product can be evaluated and certified.

The vulnerabilities in an SoC can be widely divided into three level of abstraction: *Component, Subsystem*, and *System (or SoC)*. While component level and SoC level vulnerabilities are self-explanatory, subsystem level vulnerabilities are those which arise from a composition of two or more components. The problem of designing a secure composition of several trusted or untrusted components remains an open research problem for several decades now.

An adversary has also several ways to exploit these vulnerabilities based on available resources and applications. As mentioned before, these exploit can be triggered through any effective combination of hardware, software, and interconnect level methods. For example, an AES accelerator, a component in $\mathbb{H} \cap \mathbb{TCB}$, providing confidentiality and integrity to different components of the SoC can itself be vulnerable to power-based side-channel attacks. This is component level vulnerability exploited at hardware-level and would require integration of side-channel countermeasures like masking. Similarly, a PUF-based key management unit is a subsystem. An adversary with fault injection capability can alter the PUF response leading to subsystem level vulnerability resulting in denial of service (DoS) attack which can be triggered either by external equipment like laser or remotely through software using features like dynamic voltage scaling. Furthermore, open debug interfaces ($\mathbb{N}$) can provide an attacker unrestricted access to internal components resulting in a system level vulnerability.

In the following, we discuss the threat model with respect to ITUS. ITUS is a RISC-V-based SoC designed for security-oriented embedded applications in physically hostile environment. It is expected to resist an attacker with the following capabilities:

– Physical access to external ports including test/debug ports further triggering privilege escalations through software [19].
– Can read external memory through physical probing, debug ports, or software privilege escalation [20].
– Can inject faults [21] into one or several components (ex. AES, PUF, TRNG) through external means like laser and glitching etc., or software [22].
– Can access side-channel leakage through physical means like power, EM [23], or software methods like cache-timing and speculative execution [19, 20].
– Can exploit existing vulnerabilities in the OS / $\mathbb{S}_S$.

While the protection mechanisms in the current version of ITUS do not yet cover all the threats in the comprehensive target threat model described above, the following sections discuss the threat coverage due to the various building blocks implemented so far.

3 Building Blocks of a Secure SoC

This section outlines some important building blocks ($\in \mathbb{TCB}$) of a secure SoC under three groups: (1) Securing Roots-of-Trust, (2) Micro-architectural Security, and (3) Trusted Execution Environment. Implementation choices depend on the presumed threat model, performance requirements, and resource constraints. Implementation details of these, in ITUS, will be discussed in Section 4 together with their role in the larger security subsystems.

3.1 Hardware Roots-of-Trust

Root-of-trust (RoT), a term used with many connotations, could be understood in two ways:

– RoT, fundamentally, as a relative notion—i.e., "trust in A" is derived from B' $\implies$ 'B is a RoT of A .

– RoTs as distinct subsystems offering a range of security services to a host/main system.

The fundamental RoTs of a secure SoC are the device secret keys (perhaps derived from a PUF, or separately provisioned) and their certifications as all members of $\mathbb{TCB}$ ultimately derive their trust from them ($\text{RoT}_{0,\text{hardware}}$). For the software stack, the zero-stage bootloader could be considered the smallest RoT ($\text{RoT}_{0,\text{software}}$). The RoTs are usually implemented $\in \mathbb{H}_E$—e.g., TPM [24], DesignWare tRoot Vx HSMs, Rambus CryptoManager RoT, Google Titan, Microsoft Cerberus (a hierarchical RoT structure).

Some of these RoTs could also be implemented within a host SoC chip ($\in \mathbb{H}_I$), but as distinct subsystems, e.g., OpenTitan silicon RoT, Rambus VaultIP RoT. These are small footprint resilient embedded systems that contain and offer, chiefly, the services described in Sections 3.1.1, 3.1.2, and 4.1.

3.1.1 Cryptographic Primitives

While physical security measures, from access control policies down to materials, form the first line of defense, the sole other means to enforce a set of security policies is through protocols built out of cryptographic primitives. For reasons of performance and security, these are implemented as specialized co-processors or as dedicated hardware accelerators with controlled access.

Private-Key Cryptography: Used to ensure data encryption and integrity during storage, transmission, and even prevent reverse engineering. To prevent a few side-channel attacks, circuit-level techniques [25] and encoding schemes [26] are used, and these incur non-negligible chip real-estate. These schemes are not known to be quantum-unsafe. Lightweight alternatives are available.

Public-Key Cryptography: Used in setting up authentication protocols such as in device and firmware provisioning, secure boot, and software updates. However, robustness of these implementations against stealthy side-channel attacks is a major challenge (e.g., [27, 28]). Many classical schemes are known to be quantum-unsafe. There has been active recent work evaluating industrial use-cases of post-quantum cryptography primitives [29] and also some hardware studies within RISC-V SoCs: pqsoc.com, [16, 30].

Unkeyed Primitives: This class includes hash/one-way functions. TRNG and PUF (a physical one-way function) primitives (Section 3.1.2) may also be placed under this class.

3.1.2 Key Management

The security of public-/private-key primitives relies on secure key management. Key management in a secure SoC covers functions such as secure key generation (including using unbiased entropy sources), distribution, storage, and aiding key-based operations such as identification, authentication, and cryptographic measurements/attestations as needed in an RoT. While secure key storage in conventional key managers contains non-volatile memories with protections such as tamper-proof or tamper-evident packaging, the use of on-chip entropy sources to generate keys on-demand obviates the need for key storage; however, there is still a need for some one-time programmable (OTP) storage to store certifications. Traditionally, this is done through RoT chips such as TPM.

TRNG and PUF: A True Random Number Generator (TRNG) and a Physical Unclonable Function (PUF) in a digital system extract entropy from diverse physical phenomena such as thermal noise and clock drift. Beyond key management, a TRNG finds uses in many components of the $\mathbb{TCB}$ as well as in construction of some countermeasures (e.g., masking). A PUF [31] enables lightweight and secure key management. Biases are inherent in their constructions and a post-processing step (a "whitening" process) is usually applied. Active attacks leading to denial-of-service (DoS) or introduction of biases, either through fault injection or even hardware Trojans, are an attack vector.

3.2 Core Micro-architectural Security

Traditional software enforced security depends on some basic mechanisms part of all modern processor specifications, e.g., the privilege modes *M, S*, and *U*. Security-related additions to ISA specifications (such as the PMP and IOPMP primitives in RISC-V) support building portable trust solutions in $\mathbb{S}$. Most privilege escalation vulnerabilities are due to security bugs in $\mathbb{S}$ (e.g., memory safety bugs), and hardware-assisted countermeasures have been proposed to catch them (e.g., control-flow integrity, protected pointers). However, even a bug-free $\mathbb{S}$ can be vulnerable to several side/covert channels [32] made possible due to how the processor micro-architecture is implemented. Even the safe enclaves offered by TEE frameworks can be vulnerable to side-channel leaks such as due to speculative execution [33]. Measures to ensure fault resiliency are another important aspect in this context, e.g., DIVA [34], Sentry [35].

3.3 Trusted Execution Environment

A trusted execution environment (TEE) [36] can be seen as an extension of capabilities in the secure execution services offered by the discrete RoTs, under the TCB of an SoC. A program executing in a TEE is said to enclaved. A TEE offers an enclave ($\in \mathbb{S}_U$) isolation guarantees on untrusted or vulnerable shared resources (e.g., the OS /$\mathbb{S}_S$, main memory, caches). This protects the enclave from security incursions (e.g., confidentiality, integrity) from untrusted components in $\mathbb{S}_U \cup \mathbb{S}_S$—or even the other way around, depending on the threat models. The enclaved program may seek attestation certificates on the TCB of the platform to verify its trust and the enclaved program itself is measured to ensure it was not tampered during load. A TEE framework builds on the hardware root-of-trust as described in Section 3.1. TEEs also need side-/covert-channel CMs [32]. Examples of TEE frameworks—RISC-V: Sanctum : RISC-V—Sanctum [10], Keystone [11], TIMBER-V [12]; Others: Intel Others—Intel SGX, AMD SEV, ARM TrustZone.

4 Security Subsystems

This section presents the security subsystems built using specific implementations of the blocks described in the previous section. Excluding Section 5 (which is evaluated through RTL simulation of an out-of-order RISC-V core [37]), the other components are part of ITUS RISC-V SoC based on the low RISC project with Rocketchip RISC-V, Fig. 1 and have been evaluated on Xilinx Kintex 705 FPGA board.

4.1 Secure Boot

Secure boot, a fundamental protocol in a secure SoC, has the responsibility to build a "chain-of-trust" (CoT) as it verifies the integrity of components of the TCB during the boot process. The chain starts from a root-of-trust (RoT_0, Section 3.1) of the TCB. The boot process is aborted if the verification of any stage fails; otherwise, the system is expected to be running in a trusted state.

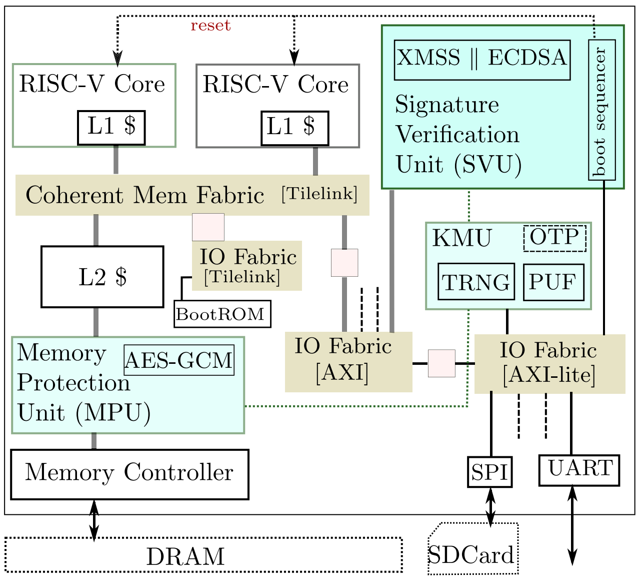

Fig. 1 The ITUS Secure SoC

This definition of secure boot is widely accepted in the security community [38]. The responsibility of maintaining the integrity of the booted system, against time-of-check to time-of-use (TOCTOU) attacks, rests with the other security mechanisms such as in Sections 4.3 and 4.2. For resource-constrained devices, private-key cryptography is used for integrity checks (e.g., the automotive SHE [39] module); however, secure management and provisioning of keys and device firmware are more practical with public key-based signature schemes.

Figure 2 illustrates a sketch of the boot sequence building the CoT as ITUS boots . At power-on, the $\mathrm{RoT}_{0,\mathrm{hardware}}$ is validated (device key generation and their verification of their certification). The subsequent stages in the CoT are the various bootloader stages ($\in \mathbb{S} \cap \mathbb{TCB}$). The public-key (PK) signature on the computed hash-digest of each stage is verified before moving to the next; the stage-wise hash-digests are hash-extended ($A \leftarrow \mathrm{hash}(A_{\mathrm{old}}||B)$) into a hardware register as a record, used to attest to integrity of the CoT (or TCB) to a local/remote client. For the zero-stage bootloader (ZSBL), the PK needs to be signed by a trusted party whose PK is stored in an on-chip one-time-programmable (OTP) memory.

4.1.1 Implementation Details

The signature verification unit (SVU), a hardware module in Fig. 1, has been integrated as a memory-mapped peripheral in the ITUS SoC to support the secure boot process (and for potentially other uses during the lifetime of the device). Two signature scheme options have been evaluated—(1) the classical Elliptic Curve digital signature algorithm (ECDSA), reported in [15]; (2) the post-quantum eXtended Merkle signature scheme (XMSS), reported in [16]. Both ECDSA and XMSS verification modules are implemented fully as hardware.

The first instruction, a RISC-V core (say core_0), executes after power-on is from a boot ROM, located at the reset address (32′h0). This zero-stage bootloader (ZSBL) has instructions to jump to the first-stage BL located either in an on-chip BRAM or in external memory. The ZSBL also carries as a payload a device tree structure (here, encoded as a string) that describes the peripheral devices, the memory

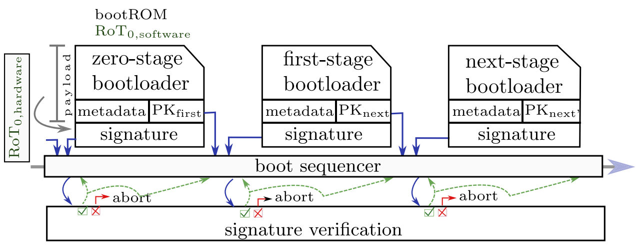

Fig. 2 Secure boot chain-of-trustFL

Table 1 Secure boot: area, latency (verification time, cycles)

Secure boot with ___	Slice LUTs	FFs	Latency
ECDSA (sect233r1) including SHA3 [43]	27170 ($\times 1 = a$)	6722 ($\times 1 = b$)	6720 ($\times 1 = c$)
XMSS (version 1)	a ($\times 2.3$)	b ($\times 2.44$)	c ($\times 25.0$)
XMSS (version 2)	a ($\times 1.26$)	b ($\times 4.16$)	c ($\times 11.1$)

map, and the cores and their capabilities. The first-stage BL (FSBL) is a small program with limited capabilities and is stored in an on-chip BRAM. It is able to locate a specific file (`boot.bin`) in the first partition on the SD card. This file contains the second and third stage bootloaders, the Berkeley bootloader (BBL)—which also contains the Security Monitor (SM) mentioned in Section 4.2—and the linux kernel (`vmlinux`) respectively. The control is transfered to BBL which then transfers control to the linux kernel, while it continues executing at the machine level to serve exceptions, traps, and as SM.

In conventional secure boot flows, the signature authentication algorithms are implemented in software ($\in \mathbb{S}_M$), e.g., SHA3 and ECDSA, executing on RISC-V such as in Sanctum [40] and Keystone [11]. In ITUS, the authentication is fully delegated to dedicated hardware units (the Boot Sequencer and the SVU). The main relative advantages of a fully hardware implementation (in this case of SHA3, ECDSA, XMSS, etc.,) are as follows: (1) performance and energy-efficiency, and considering its general usage frequency in a secure SoC; (2) helps to reduce the size of $\mathbb{S} \cap \mathbb{TCB}$, with the knowledge that attack surface for hardware is smaller.

A note on attack vectors on secure boot: Fault-based bypass attacks are common and very practical [41, 42] AVs on secure boot. Countermeasures involve using redundancy in space and time (with randomization)—making the attacks hard to target. For secure boot, detection-based countermeasures suffice as the boot can be aborted once a fault attack is detected (as "availability" is not a concern at this stage). Memory spoofing attack post a successful secure boot is also an AV and can be detected through memory integrity checks such as using the MPU (Section 4.3) managed memory.

4.1.2 Evaluation

While more details are reported in [15, 16], the key observations are summarized here. Tables 1 and 2 compare secure boot (SVU + signature scheme) resource usage and latency (to verify ZSBL). Assuming a conservative 100 MHz clock, the verification times range between 67 μs and 1.6 ms—sufficient for most systems. Both these schemes (XMSS, ECDSA) are able to re-use significant portion of the hardware instantiated for verification operation for their corresponding signature and key generation operations (which are often more computationally expensive). This makes the hardware implementation of these modules worthwhile considering their use-cases in the lifetime of a secure SoC.

4.2 Secure Enclaves: Integrating Keystone

ITUS integrates the open source framework, Keystone [11], for building customizable TEEs (see Fig. 3). Keystone depends the physical memory protection (PMP) priv-1.10 specification of RISC-V to support memory isolation for its enclaves.

The reader is referred to [11] for more details on capabilities and threat coverage offered by Keystone. This section re-states some essential details in order to discuss its integration with ITUS. In Fig. 3, Enclave$_1$ is an instance of Keystone-based enclave running an enclave application (EA) in $\mathbb{S}_U$. Every enclave has its own runtime (RT) which, however, belongs in $\mathbb{S}_S$. RT and EA share the same enclave virtual address space (EVAS), which is managed by the RT. The physical address region corresponding to this EVAS is isolated with enforcement using PMP configuration. The untrusted components of $\mathbb{S}$ outside of the enclave do not have access to this EVAS (including its page-tables). The

Table 2 ECDSA (binary curve sect233r1) implementation

	Baseline SW Intel IVB	Vectorized SW Intel IVB+AVX	Our HW FPGA
Cycles	2,226,927	405,330	6,720
Frequency (MHz)	2,000	2,000	100
Time(ms)	1.11	0.22	0.067
Power (mW)	2,100	2,600	386
Energy (mJ)	2,331	572	0.025
Energy efficiency (mJ $\times$ ms)	2,587.4	125.84	0.0016

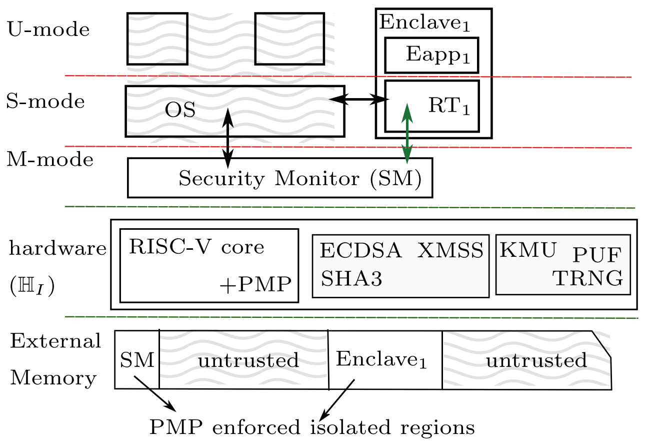

Fig. 3 ITUS SoC and Keystone (Fig. based on [11])

security monitor (SM) executes in M-mode and is part of the TCB. The address space used by SM has its own PMP isolated region. SM is responsible to provide security enforcement guarantees (runtime configuration of PMP during enclave life-cycle), cryptographic computations such as for measurements and attestations. SM exposes limited API through SBI to the RT and the Linux kernel.

An EA needs to check the attestation to the TEE framework's integrity. This attestation record could be obtained as a part of the ITUS secure boot (Section 4.1) flow.

As the enclave is created, the enclave memory space is measured to ensure the EA-related binaries are loaded correctly by the OS. This involves a one-time computation cost of hash-extending the enclave memory space and can be accelerated by using the existing hashing modules in hardware.

The current state of integration with ITUS and the planned tighter integration is inline with one of the design goals as described for Keystone [11]. It helps to reduce the size of $\mathbb{S}_M \cap \mathbb{TCB}$ by leveraging other security subsystems developed for ITUS.

4.3 Memory Protection Unit

Solutions such as a TEE framework (Section 4.2) do not offer a complete threat coverage when dealing with untrusted external storage. ITUS includes a Memory Protection Unit (MPU), reported in [18], that offers confidentiality and integrity properties on data in external memory. To minimize the performance overhead, selective regions can be marked as secure. Further support for dynamic and reconfigurable allocation of such regions is feasible. The DRAM allocation is freed upon secure boot and then repartitioned into non-trusted, trusted, and metadata regions. In compliance with the secure and trusted boot-up sequence, such allocation of regions is protected from interception and modification by software. Data reads and writes to the trusted region are processed through

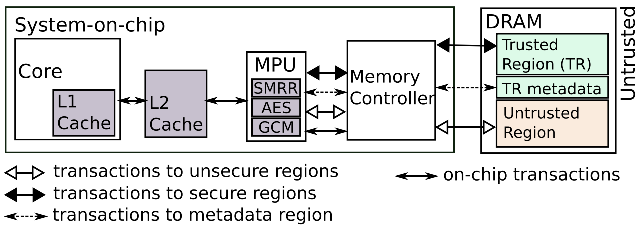

Fig. 4 Memory Protection Unit [18]

the MPU for authenticated encryption prior to and after the encapsulation. This preserves the confidentiality and integrity of sensitive data in the DRAM and at the same time guarantees minimal leakage of the address/location mapping of the memory data. In its implementation in ITUS (Fig. 4), AES-GCM is used to achieve confidentiality and integrity. The memory range partitioning is tracked by Secure Memory Range Registers (SMRR). The proposed MPU for ITUS has two main characteristics: (1) the authenticated encryption implemented through AES-GCM offers confidentially and integrity protections with the minimal usage of hardware resources; (2) a simplified integrity tree using Bonsai Merkle Tree (BMT) structure is proposed as a measure to preserve the freshness of memory contents and to thwart replay attacks.

The current MPU is vulnerable to micro-architectural side-channels (e.g., LLC). It may be noted that if Keystone were fortified through micro-architectural side-channel attack countermeasures, ensuring its isolation guarantees are strengthened, one could omit BMT and associated overheads.

Beyond the common scheme used in MPU, other countermeasure techniques such as Address Space Layout Randomization (ASLR) and Oblivious RAM (ORAM), have been reported.

Evaluation: The AES-GCM module in the MPU [18] consumes 4205 slice LUTs. The implementation requires storing only a 64 bytes of meta data on-chip (root of the integrity tree) and only about 0.43% overhead in off-chip storage.

4.4 Key Management Unit

The key management unit (KMU) as implemented in ITUS is shown in Fig. 5 and integrated as shown in Fig. 1. ITUS uses the configurable response-length LFSR-based PUF, reported in [17], with a lightweight and reliable key extraction through sequential use of BCH error correcting code BCH(15, 7, 2) operating on 7-bit segments at a time. The component PUF, shown in Fig. 5, packages the raw PUF together with BCH encoder and decoder modules to reliably extract a key string from raw PUF responses for a given challenge. It operates in two modes: in mode-1, it generates the syndrome for a given challenge string (which are to

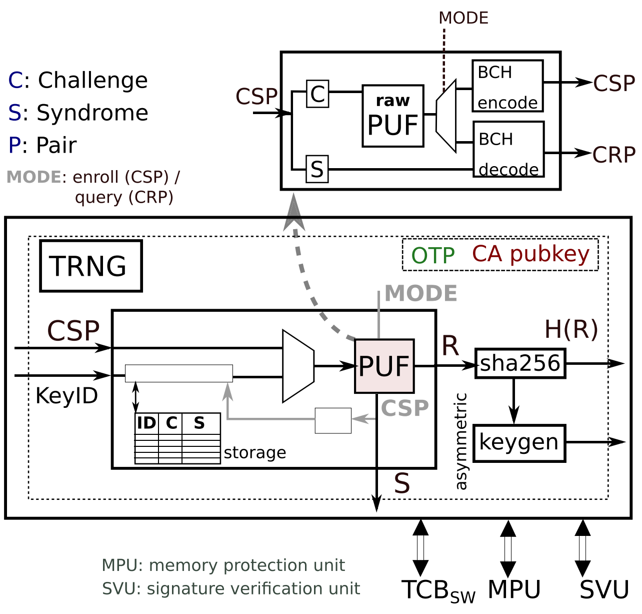

Fig. 5 PUF-based KMU

be remembered/stored) and in mode-2, given a challenge and the corresponding syndrome, the corresponding unique response is regenerated. While the error correction[2] and other functions of the KMU may also be delegated to software ($\in \mathbb{S}_M \subset \mathbb{TCB}$) depending on the threat model, it is preferable and inexpensive to implement this module completely in hardware.

KMU (Fig. 5) is integrated as a memory-mapped peripheral, with access only to the security monitor ($\in \mathbb{S}_M$), and also connected to MPU and SVU over dedicated point-to-point channels. PUF as shown in Fig. 5 operates in two modes. In one mode, it takes as input either a KeyID or a challenge-syndrome pair and generates the corresponding PUF response. KeyID refers to the handle for a PUF response for a corresponding challenge string. In the other mode (e.g., at power-on or on-demand), the ID-challenge-syndrome table is populated for a chosen set of challenges (passed through C portion of the CSP port) that are either user supplied or sourced from the TRNG, which is a simple RO TRNG based on [44]. The PUF response is whitened by hashing it with `sha256`.

The responses from PUF, corresponding to given CSPs, form the seed material from which symmetric or asymmetric keys are derived. Key derivation functions like Ed25519 can further be used to obtain keys of required strength/length. The ITUS KMU presently generates elliptic curve (EC) (sect233r1) key pairs (S_K, P_K) with secret key S_K chosen as the hashed PUF response and public key P_K is computed as the point multiplication of S_K with the public base point of the curve.

[2]e.g., BCH(398, 128, 32) capable of correcting up to 32 bit errors using 270 bits of helper data

The PUF-based KMUs undergo an enrollment phase followed by provisioning the OTP storage certifying the device (for devices without PUF, the device keys are provisioned). As this work in its present form targets FPGA, the following is one way to do this process for an FPGA. Both the ZSBL and FSBL—the former implemented as a bootrom and the latter implemented through an on-chip blockram—are part of the bitstream/design. Both ZSBL and FSBL are signed (using one of two public key signature schemes) and the signatures on them are verified in hardware without the involvement of software/cpu as a part of our secure boot protocol. As FSBL is mutable (edit the contents of BRAM in the bitstream) as long as it is appropriately signed, it would be the place where a CSP_{dev} corresponding to the device root key (to be used with the PUF-based KMU to re-generate the root keys) is placed. As for the signer's public key (important to verify signer), it is to be stored in the eFUSES on the FPGA (or alternatively, some recent FPGAs support battery backed BRAM appropriate for this purpose)—the FPGA used, KC705, does have eFUSES, but since they are one-time programmable, we omit actually doing this. For practical purposes, the signer's public key could be stored in the ZSBL instead. Fault-injection attacks here are indeed possible. They could lead to a DoS attack (hard to prevent; if injected during verification) or Signature forgery attack (if the attacker is able to modify the signer's public key; preventable by also storing a section of hash of the public key).

4.4.1 Evaluation

The PUF [17] used in KMU (Table 3) was configured to accept 16 bit challenge strings to generate 128 bit raw responses. The key extraction uses BCH(15,7,2) encoder/decoder modules.

4.5 Physical Access Management

Hardware debug instruments are circuits added to the design of a system-on-chip (SoC) to allow post-silicon debugging of the chip. Hardware debug allow the debugger

Table 3 KMU resource usage (with sub-components)

		Slice LUTs	FFs
	ECkeygen	21473	249
	PUF	6965	2027
	BCHenc	71	333
	BCHdec	251	371
	PUFraw	5554	364
KMU		29529	3344

to capture real-time performance statistics, observe and modify values of internal configuration registers, and examine the instructions execution [45, 46]. Examples of debug instruments used in commodity SoCs are internal and boundary scan chains, hardware performance counters, and JTAG [47]. The ability of debug hardware to observe the internal components of the SoC can be used as a backdoor for attacks. For example, Yang et al. [48] and Chiu et al. [49] show how an attacker can use the boundary scan chains to leak SoC secure keys.

Due to the high complexity of modern SoCs and the importance of security, a balance between security of assets and observability of a debug infrastructure should be made. To guarantee high security level, we can permanently disable the debug infrastructure after the silicon validation process is completed, for example, by blowing fuses. However, this approach violates the late variability and reusability requirements of modern SoC because trace-based debugging is needed throughout the SoC life cycle [46]. A better approach is to consider a secure debug port approach [47, 50]. The main idea is to lock the debug access port (e.g., JTAG) and to only allow trusted debuggers to access the debug infrastructure after successful authentication. This approach provides either complete or no access to debugging.

A more practical approach is to build a multi-level secure debug [51, 52]. The multi-level privilege system allows for in the field updates, and debugging of the firmware while maintaining a high degree of protection for the most sensitive intellectual property in the SoC. All security privileges can be set dynamically by the developer post-manufacturing.

The current ITUS prototype does not include the ideas in this section, however, consider the following example. Debug ports (including JTAG) are often accessible from within software/firmware, as well as externally. A "first-instruction-integrity secure boot," as is our implementation, ensures the former access path is authenticated. The latter physical access path needs to be secured through a separate authentication scheme. Instead of discussing this case for JTAG, let us consider for simplicity, the case of UART port, for which the following protocol can be implemented in software (M-mode UART specific trap function). The scenario would be that of the UART port, beyond a point during the boot flow, not allowing any I/O access over it except to offer an authentication protocol like the following.

A Simple Protocol Between the "Device" and a "Debug Host": A debug host acquires a $K_d <= \mathrm{PUF}(C_d)$ for a challenge C_d and stores this K_d securely and device remembers C_d. A secure channel to do this over would be a properly signed FSBL and secure boot. To authenticate the debug host again to grant access, the device challenges the host for a specific response. The device does $X <= \mathrm{TRNG}$; and computes $R_1 <= \mathrm{HMAC}(X, \mathrm{PUF}(C_d))$, a hash-based message authentication code, and sends X to host. The host computes $R_2 <= \mathrm{HMAC}(X, K_d)$ and sends it to device. The device grants access if $R_1 == R_2$.

5 Protection Mechanisms Against Side-Channel Attacks

Modern SoC architectures are built to support a wide range of usage models and applications, and they strive to achieve the best power and performance. To achieve this goal, systems optimize the execution of "common execution paths" and widely use different speculative micro-architectural mechanisms such as prefetching, out-of-order execution, value prediction, and more. These mechanisms have been shown [53–55] to leak information side-channels leading to successful attacks. However, since these mechanisms are essential to cope with the growing demand for computing power, most SoC providers prefer to compromise security over giving up power or performance. This section tries to address the three important observations we make regarding the security of SoC devices: (1) protecting current complex systems has become a major challenge, (2) the current proposed solutions are too complicated, require too much power, and/or come with major power and performance overheads, and (3) there is no effective mechanism to protect the system against future, unknown attacks—since many SoC systems, such as for the automobile industry, presume to serve the products for 15–20 years, it is unlikely that using existing security vectors will be sufficient for the entire life of the such SoCs. Thus, we believe that a new approach is required to meet these new challenges of securing future SoC architectures.

In particular, we suggest the following two techniques: (1) Introducing a new "security wrapper" that aims to provide another abstraction layer to the implementation; (2) Using Dynamic Security Protection (DsP)-based mechanisms. These are discussed in Sections 5.1 and 5.2, followed by an example applying them to protect a system against side-channel timing attacks in Section 5.3.

5.1 The Security Wrapper

In order to build a secure system without causing a significant power or performance overhead, we propose to add a security wrapper to the implementation. This wrapper aims to separate actual "events" from the way we expose it to the user. For example, to protect a system against side-channel timing attacks, we present a different view of timing events (such as commit times) to the user/attacker from the actual (based on execution times). The suggested mechanism resembles the way modern architectures implement out-of-order mechanisms, wherein, instructions are fetched and committed in the program's order, allowing the user/tools to take advantage of a simpler abstraction. But internally, the system executes instructions out-of-order (dataflow style) and speculatively, allowing it keep to the performance advantages. The security wrapper (see Fig. 6) aims to achieve a similar goal: to present the user a secure view on the way the system is executing, while allowing the internal implementation to mainly focus on the performance and power aspects, and worry less about their impact on security.

The proposed new architecture aims to provide the external world with a view that is different from the actual execution characteristics and with minimal impact on performance. As the attacker of the system can only observe the "secure core," the system must impose restrictions only to make sure this abstraction is maintained, even if the underlying architecture may not support it. Thus, the internal core can execute code in a non-secure manner, as long as an external observer gets a view of a secure core.

Attack vectors change and evolve together with the threat model and further differ between different products and also with newer attacks discovered over the lifetime of a project. In order to dynamically tailor or maintain the secure view of the system, we introduce the second principle of our approach: *Dynamic Security Protection (DsP)* mechanism.

5.2 Dynamic Security Protection-Based Mechanisms

This technique is motivated by the way functional bugs are handled in modern systems to avoid risking the prospect of a mass recall of systems:

– Inserting hooks in the system with capabilities to detect, bypass, and even resolve future bugs, e.g., many Intel architectures include a mechanism for "dead-lock breaker" that is able to (1) detect if a dead-lock occurs and (2) restart the system from a stable point when it happens.
– All complicated sequences that are tuned to functional bugs are controlled by microcode and so can be modified if a new bug found.

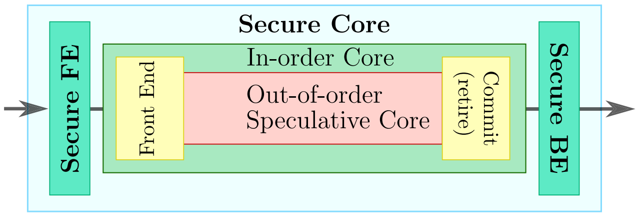

Fig. 6 High-level structure of a security wrapper of a core

– Bugs and malfunction behaviors are allowed as long as they are not exposed to the outside world, e.g., a single-bit flip of data is allowed as long as an error detection and correction mechanism exists that can correct the value during the read or write operation.
– Introducing redundancy at certain points of the system to allow resolution of "malfunction behaviors."

Adopting these principles to handle security attacks, we suggest:

To consider all possible attack vectors (even attack vectors that are not relevant to our product) and to add mechanisms to (1) detect if such an attack happened and (2) to help block the information leaks to the external world if such an attack happens. Please note that if such mechanisms were to cause performance or power overhead, we may decide to operate them only for products that consider this attack as part of their threat model.

To add a controller that controls the detection and reaction sequence. Such a controller should be programmable, similar to microcode, so it could be changed from one product to another, could be adjusted to the needs and limitations of the product, and could be used to address future requirements.

To consider having open interfaces for future sensors that could be added, similar to accelerators, if needed. This technique can be very efficient for the automotive industry with 15–20-year product life.

5.3 Put It All Together—an Example: Protecting Against Side-Channel Timing Attacks

This section demonstrates the newly proposed technique by providing a detailed example of how it can be used to protect a modern out-of-order (OOO) system against timing side-channel attacks. Side-channel timing attacks can occur if the time it takes to complete the execution of a code segment depends on the value it manipulates (A.K.A a "secret"). Countermeasures against timing attacks are divided between software-based and hardware-based techniques [56]. Among the Software techniques to protect against timing side-channel attack is to pad the code with NOPs of other "dummy instructions" so that the execution time of all execution paths is balanced. The hardware-based techniques add extra logic to balance the different execution paths, so they are not dependent on the values being computed. This task can be costly. The software techniques require intensive work from the programmer and must be re-balanced from one architecture to another. The hardware implementations can be costly in terms of performance and power, especially if they are implemented for general-purpose execution units or require the use of

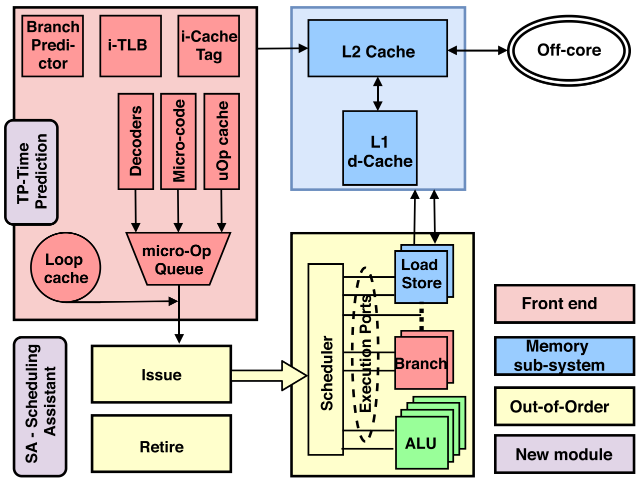

Fig. 7 Core Internals. *Time Prediction* (TP)—Measure execution time of a code segment. Indicates how much time the commit needs to be postponed (if current execution time is shorter than the previous time) or record the new execution time (otherwise). *Scheduling Assistant* (SA)—Extends the functionality retirement (commit) mechanism by delaying the time, the instruction will be committed

separate execution units for security purposes (e.g., AES instruction set extension [57]).

The alternative solution presented here is based on the observation that a user can only measure time with respect to the time instructions are committed, while the overall performance mainly depends on the time the instructions were actually executed. Thus, we propose a mechanism that delays the commit time, while keeping the execution time untouched (see Fig. 7). For this example:

For the **Security wrapper**, we will add: (1) a mechanism to indicate "begin" and "end" markers of the "secure section'; (2) a Time Prediction unit (Fig. 7); (3) a Scheduling Assistant unit (Fig. 7).

For the **Dynamic Security Protection**, we will add a control unit that can be implemented as part of the time prediction unit or as an independent controller. This unit will be controlled by microcode that could update its algorithm if needed or when new class of attacks were to be discovered. This controller may be different from one product to another. For example, for some products, it may support only a single security segment at a time, for other products, it may contain a cache that can support multiple security regions and even nested regions.

5.4 A Proof-of-Concept with an OoO RISC-V Core

In order to check our basic assumption that delaying a commit time has an only a minor impact on the overall performance of the system, we modified the architecture of the RISC-V-based out-of-order core, named Riscy-OOO [37] and introduced a delay to instructions, ready to be committed. We incorporate the delay logic within the Riscy-OOO design and run it using different amount of delay (for all ALU and FPU instructions.) The configuration parameters used for this experiment and observations are in Table 4.

Table 4 indicates that, for most of benchmarks, delaying ALL instructions with a small number of cycles has almost no effect on performance. More than that, other experiments we conducted showed that when increasing the size of the ROB, much smaller performance loss was measured even for larger delays. Please note that in our case, we are deferring only instructions belonging to the secure paths and so, one may assume that no noticeable performance or power loss will be imposed on the system (also, delaying the commit time causes no power implications). It may be noted that Table 4 provides sensitivity analysis of the impact of delayed commits on the performance, to motivate the fact that the impact is insignificant. As for the security analysis, at this stage, we offer it "by construction" of the countermeasure. A proper security analysis would be possible after we implement the TP and SA mechanism indicated in Fig. 7. Note also that we do not claim the proposed strategy works if speculation operations are involved, e.g., with speculative loads/stores, we need other ways to mask their effects.

TimeWarp [58] is a related work that proposes to fudge the timestamp counter, used to measure time, by delaying

Table 4 Impact of delaying all instruction commits in the benchmark programs in terms of %-change in execution cycles (column heads: delay amount)

	4	8	16	32	64	128	256
dhrystone	0	0.1	-0.1	10.2	52.3	162.1	381.9
multiply	0	0.1	0.2	3.0	36.0	134.2	347.7
qsort	0	0	0.4	0.6	0.2	8.9	66.3
rsort	0	0	0.1	27.1	101.3	266.5	604.7
spmv	0.1	0.2	-0.5	2.6	30.0	113.1	303.0

RiscyOOO configuration: Superscalarity: 2; Branch predictor: Tournament; Size reservation-station (RS): 16; RS for FPU ops: 16; RS for memory ops: 16; Reorder-buffer: 64 entries; L1 Cache: Size 32kB, Ways 8; LLC: Size 1 MB, Ways 16; DRAM latency: 120 cycles (constant); BootROM: 4kB the commit of the `rdtsc` instructions (system wide or within a process). It is a technique that can break some critical software and also makes assumptions about how fine the virtual timestamp counters can be constructed.

5.4.1 A Motivating Example:

The classic modular exponentiation could be one example but let us consider the following case instead:

```
  // message authentication code
compute+compare
START_SECURE;
compare_unsafe(MAC(key, msg), tag_256 bytes);
END_SECURE;
```

If $\texttt{compare}_{\texttt{unsafe}}$ is implemented as a byte-by-byte compare in a loop, exiting early on a mismatched byte, it is unsafe (attacker would be able to easily acquire a valid tag for any message by measuring time). One countermeasure is to implement `compare` as constant time, but such cost is not necessary. If the TP/SA units choose instruction commit delays such that if the attacker is only able to measure $t_{\text{END}} - t_{\text{START}}$ as a uniformly random value between 1 and 256 ($\times$ a constant), we would have perfectly decorrelated data from timing, nullifying the attack.

5.5 Conclusions and Discussion

This section presents a new design for security methodology that allows considering security as a first-class citizen among other design considerations. We demonstrate some early evidence of the methodology for securing an out-of-order architecture against side-channel timing attacks, but we believe that the methodology can be extended to other security domains as well.

6 Related works

While the development of secure implementations of the basic primitives discussed in Section 3, including countermeasures for the many perceived threat models, has a much longer history (the bottom-up approach), there have increasingly been significant contributions towards building secure systems (top-down) from both commercial and academic quarters.

Examples of these secure systems, of a smaller scale, are known as Roots-of-trust modules, examples of which are listed in Section 3.1. A recent notable system in this category is OpenTitan silicon RoT[3], built around a small RISC-V core (Ibex).

Examples of mainstream SoCs with hardware-assisted security features include ARM TrustZone, Intel SGX, AMD SEV. A few newer RISC-V-based commercial SoCs also join this list, e.g., the SiFive Shield, Hex-Five Security (MultiZone secure domains). These security features are essentially TEE frameworks that guarantee different security properties. Academic projects offering TEE solutions include Sanctum [10], Keystone [11], Timber-V [12], MI6 [33], and AEGIS [9]. [36] discusses some of these TEEs. AEGIS is a process to build computing systems secure against physical and software attacks. In the threat model, all the components external to the processor, such as memory, are considered to be untrusted. AEGIS provides a tamper-evident, authenticated environments in which any physical or software tampering by an adversary is guaranteed to be detected. Keystone initiated an open-source project for building trusted execution environments (TEE) with secure hardware enclaves, based on the RISC-V architecture. Sanctum introduces software-based secure boot and remote attestation process. While Keystone strives to be less dependent on hardware modifications (improving from Sanctum), approaches like TIMBER-V use tagged memory ideas to implement flexible isolation and involve significant hardware changes[4]. Different in approach and scope from the above containment strategies, TrustGuard [35] presents a hardware-based containment strategy to "quarantine" any faulty or malicious behavior of untrusted components using a gate-keeping hardware module they call a "Sentry."

Also receiving attention, and something with a significant impact on the threat surface of a system, are CPU micro-architectural countermeasures, e.g., InvisiSpec [59] (making speculation-associated leakage unobservable), [33] (securing enclaves executing on a speculative CPU, same as the one used in Section 5.4).

7 Conclusion

The manuscript presented ITUS, a RISC-V-based secure SoC prototype that integrates a number of security subsystems (Section 1.1). While the current version of ITUS has only partial threat protection coverage with respect to the comprehensive target threat model set out in Section 2, it sets a stage for exploration of more countermeasure strategies (such as in Section 5); development of a security monitoring/validation methodology to detect security policy violations through simulation/emulation; and, working towards more effective integration of available countermeasures.

[3]https://opentitan.org/

[4]6.25% overhead, 2 bit-tag over every 32-bit word

Funding The authors acknowledge the support from the Singapore National Research Foundation ("SOCure" grant NRF2018NCR-NCR002-0001 - https://www.green-ic.org/socure)

References

1. Kim Y, Daly R, Kim J, Lee JH, Lee D, Wilkerson C, Lai K, Mutlu O (2014) Flipping bits in memory without accessing them: an experimental study of dram disturbance errors. In: Proceeding of the 41st Annual International Symposium on Computer Architecuture, ser. ISCA '14. IEEE Press, Piscataway, pp 361–372
2. Lipp M, Schwarz M, Gruss D, Prescher T, Haas W, Fogh A, Horn J, Mangard S, Kocher P, Genkin D, Yarom Y, Hamburg M (2018) Meltdown: reading kernel memory from user space. In: 27th USENIX Security Symposium (USENIX Security 18). Baltimore, MD: USENIX Association, pp 973–990
3. Van Bulck J, Minkin M, Weisse O, Genkin D, Kasikci B, Piessens F, Silberstein M, Wenisch TF, Yarom Y, Strackx R Foreshadow: extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In: Proceedings of the 27th USENIX Security Symposium. USENIX Association, August 2018, see also technical report Foreshadow-NG
4. Canella C, Pudukotai Dinakarrao SM, Gruss D, Khasawneh KN (2020) Evolution of defenses against transient-execution attacks. In: Proceedings of the 2020 on Great Lakes Symposium on VLSI, pp 169–174
5. Jang Y, Lee S, Kim T (2016) Breaking kernel address space layout randomization with intel tsx. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, ser. CCS '16. ACM, New York, pp 380–392. [Online]. Available: https://doi.org/10.1145/2976749.2978321
6. McIlroy R, Sevcík J, Tebbi T, Titzer BL, Verwaest T (2019) Spectre is here to stay: an analysis of side-channels and speculative execution, CoRR, [Online]. Available: arXiv:1902.05178
7. Ravi P, Najm Z, Bhasin S, Khairallah M, Gupta SS, Chattopadhyay A (2019) Security is an architectural design constraint, Microprocessors and microsystems, vol 68, pp 17–27, [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933118302229
8. Knechtel J, Kavun EB, Regazzoni F, Heuser A, Chattopadhyay A, Mukhopadhyay D, Fei Y, Belenky Y, Levi I, Güneysu T, Schaumont P, Polian I (2020) Towards secure composition of integrated circuits and electronic systems: on the role of eda
9. Suh GE, Clarke D, Gassend B, Van Dijk M, Devadas S (2003) Aegis: architecture for tamper-evident and tamper-resistant processing. In: Proceedings of the 17th annual international conference on Supercomputing. ACM, pp 160–171
10. Costan V, Lebedev I, Devadas S (2016) Sanctum: minimal hardware extensions for strong software isolation. In: USENIX Security Symposium, pp 857–874
11. Lee D, Kohlbrenner D, Shinde S, Asanovic K, Song D (2020) Keystone: an open framework for architecting trusted execution environments. In: Proceedings of the Fifteenth European Conference on Computer Systems, ser. EuroSys '20
12. Weiser S, Werner M, Brasser F, Malenko M, Mangard S, Sadeghi A-R (2019) Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v. In: Proceedings 2019 - Network and Distributed System Security Symposium (NDSS). Internet Society
13. Menon A, Murugan S, Rebeiro C, Gala N, Veezhinathan K (2017) Shakti-t: a risc-v processor with light weight security extensions. In: Proceedings of the Hardware and Architectural Support for Security and Privacy, ser. HASP '17. Association for Computing Machinery, New York. [Online]. Available: https://doi.org/10.1145/3092627.3092629
14. Kumar VBY, Chattopadhyay A, Haj-Yahya J, Mendelson A (2019) Itus: a secure risc-v system-on-chip. In: 2019 32nd IEEE International System-on-Chip Conference (SOCC), p 418–423
15. Haj-Yahya J, Wong MM, Pudi V, Bhasin S, Chattopadhyay A (2019) Lightweight secure-boot architecture for risc-v system-on-chip. In: 20th International Symposium on Quality Electronic Design (ISQED), pp 216–223
16. Kumar VBY, Gupta N, Chattopadhyay A, Kasper M, Krauß C, Niederhagen R (2020) Post-quantum secure boot. In: Design, Automation & Test in Europe Conference & Exhibition. IEEE, Grenoble
17. Srinivasu B, Pudi V, Chattopadhyay A, Lam K (2018) CoLPUF : a novel configurable LFSR-based PUF. In: APCCAS. IEEE, pp 358–361
18. Wong MM, Haj-Yahya J, Chattopadhyay A (2018) Smarts: Secure memory assurance of risc-v trusted soc. In: Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy, ser. HASP '18. ACM, New York, pp 6:1–6:8. [Online]. Available: https://doi.org/10.1145/3214292.3214298
19. Lipp M, Schwarz M, Gruss D, Prescher T, Haas W, Mangard S, Kocher P, Genkin D, Yarom Y, Hamburg M (2018) Meltdown, arXiv:1801.01207
20. Kocher P, Genkin D, Gruss D, Haas W, Hamburg M, Lipp M, Mangard S, Prescher T, Schwarz M, Yarom Y (2018) Spectre attacks: exploiting speculative execution, arXiv preprint:1801.01203
21. Bar-El H, Choukri H, Naccache D, Tunstall M, Whelan C (2006) The sorcerer's apprentice guide to fault attacks. Proc IEEE 94(2):370–382
22. Murdock K, Oswald D, Garcia FD, Van Bulck J, Gruss D, Piessens F (2020) Plundervolt: software-based fault injection attacks against intel sgx. In: 2020 IEEE Symposium on Security and Privacy (SP)
23. Kocher P, Jaffe J, Jun B (1999) Differential power analysis. In: Annual International Cryptology Conference. Springer, pp 388–397
24. Group TC (2011) TPM main specification level 2 version 1.2, revision 116
25. Ravi P, Bhasin S, Breier J, Chattopadhyay A (2018) Ppap and ippap: Pll-based protection against physical attacks. In: 2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), pp 620–625
26. Gupta N, Jati A, Chattopadhyay A, Sanadhya SK, Chang D (2017) Threshold implementations of gift: a trade-off analysis, Cryptology ePrint Archive, Report 2017/1040, https://eprint.iacr.org/2017/1040
27. Genkin D, Shamir A, Tromer E (2013) Rsa key extraction via low-bandwidth acoustic cryptanalysis, Cryptology ePrint Archive, Report 2013/857, https://eprint.iacr.org/2013/857
28. Bhattacharya S, Mukhopadhyay D (2016) Curious case of rowhammer: flipping secret exponent bits using timing analysis, Cryptology ePrint Archive, Report 2016/618, https://eprint.iacr.org/2016/618
29. Niederhagen R et al Industrial use cases and requirements for the deployment of post-quantum cryptography, volume wp.1, Fraunhofer Institute for Secure Information Technology, Technical Report. [Online]. Available: https://quantumrisc.org/results/quantumrisc-wp1-report.pdf
30. Fritzmann T, Sharif U, Müller-Gritschneder D, Reinbrecht C, Schlichtmann U, Sepulveda J (2019) Towards reliable and secure post-quantum co-processors based on risc-v. In: 2019 Design, Automation Test in Europe Conference Exhibition (DATE), pp 1148–1153

31. Gassend B, Clarke D, van Dijk M, Devadas S (2002) Silicon physical random functions. In: Proceedings of the 9th ACM Conference on Computer and Communications Security, ser. CCS '02. ACM, New York, pp 148–160. [Online]. Available: https://doi.org/10.1145/586110.586132
32. Szefer J (2016) Survey of microarchitectural side and covert channels, attacks, and defenses, Cryptology ePrint Archive, Report 2016/479, https://eprint.iacr.org/2016/479
33. Bourgeat T, Lebedev I, Wright A, Zhang S (2018) Arvind, and S. Devadas, MI6: secure enclaves in a speculative out-of-order processor, CoRR, [Online]. Available: arXiv:1812.09822
34. Austin TM (1999) Diva: a reliable substrate for deep submicron microarchitecture design. In: Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture, ser. MICRO 32. IEEE Computer Society, USA, p 196–207
35. Zhang H, Ghosh S, Fix J, Apostolakis S, Beard SR, Nagendra NP, Oh T, August DI (2019) Architectural support for containment-based security. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS '19. Association for Computing Machinery, New York, pp 361–377
36. Jauernig P, Sadeghi A, Stapf E (2020) Trusted execution environments: properties, applications, and challenges. IEEE Secur Privacy 18(2):56–60
37. Zhang S, Wright A, Bourgeat T (2019) Composable building blocks to open up processor design. IEEE Micro 39(3):47–55, https://github.com/csail-csg/riscy-OOO
38. Sau S, Haj-Yahya J, Wong MM, Lam KY, Chattopadhyay A (2017) Survey of secure processors. In: 2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), pp 253–260
39. Sau S (2009) SHE: secure hardware extension version 1.1
40. Lebedev I, Hogan K, Devadas S (2018) Secure boot and remote attestation in the sanctum processor, Cryptology ePrint Archive, Report 2018/427, https://eprint.iacr.org/2018/427
41. Timmers N, Spruyt A (2016) Bypassing secure boot using fault injection, Blackhat Europe 2016
42. de Haas J (2013) 20 ways past secure boot, Hack in the Box Security Conference
43. Wong MM, Haj-Yahya J, Sau S, Chattopadhyay A (2018) A new high throughput and area efficient sha-3 implementation. In: 2018 IEEE International Symposium on Circuits and Systems (ISCAS), pp 1–5
44. Wold K, Tan CH (2008) Analysis and enhancement of random number generator in fpga based on oscillator rings. In: 2008 International Conference on Reconfigurable Computing and FPGAs, pp 385–390
45. Vermeulen B, Goossens K (2014) Debugging systems-on-chip: communication-centric and abstraction-based techniques. Springer
46. Orme W (2008) Debug and trace for multicore socs, ARM White paper
47. Rosenfeld K, Karri R (2010) Attacks and defenses for jtag. IEEE Des Test Comput 27(1):36–47
48. Yang B, Wu K, Karri R (2004) Scan based side channel attack on dedicated hardware implementations of data encryption standard. In: 2004 International Conferce on Test. IEEE, pp 339–344
49. Chiu G-M, Li JC-M (2010) A secure test wrapper design against internal and boundary scan attacks for embedded cores. IEEE Trans Very Large Scale Integr (VLSI) Syst 20(1):126–134
50. Pierce L, Tragoudas S (2012) Enhanced secure architecture for joint action test group systems. IEEE Trans Very Large Scale Integr (VLSI) Syst 21(7):1342–1345
51. Pierce L (2011) Multi-level secure jtag architecture. In: 2011 IEEE 17th International On-Line Testing Symposium. IEEE, pp 208–209
52. Das A, Da Rolt J, Ghosh S, Seys S, Dupuis S, Di Natale G, Flottes M-L, Rouzeyre B, Verbauwhede I (2013) Secure jtag implementation using schnorr protocol. J Electron Test 29(2):193–209
53. Kocher P, Horn J, Fogh A, Genkin D, Gruss D, Haas W, Hamburg M, Lipp M, Mangard S, Prescher T, Schwarz M, Yarom Y (2019) Spectre attacks: exploiting speculative execution. In: 2019 IEEE Symposium on Security and Privacy (SP), pp 1–19
54. Mcilroy R, Sevcik J, Tebbi T, Titzer BL, Verwaest T (2019) Spectre is here to stay: an analysis of side-channels and speculative execution
55. Maisuradze G, Rossow C (2018) Speculose: analyzing the security implications of speculative execution in cpus
56. Ge Q, Yarom Y, Cock D, Heiser G (2018) A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. J Cryptogr Eng 8(1):1–27
57. Gueron S (2009) Intel's new aes instructions for enhanced performance and security. In: Fast software encryption, Dunkelman, O, Ed. Springer, Berlin, pp 51–66
58. Martin R, Demme J, Sethumadhavan S (2012) Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In: 2012 39th Annual International Symposium on Computer Architecture (ISCA). IEEE, pp 118–129
59. Yan M, Choi J, Skarlatos D, Morrison A, Fletcher CW, Torrellas J (2018) Invisispec: making speculative execution invisible in the cache hierarchy. In: Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO-51. IEEE Press, Piscataway, pp 428–441