*electronics*

MDPI

*Article*

# A Trusted Execution Environment RISC-V System-on-Chip Compatible with Transport Layer Security 1.3

**Binh Kieu-Do-Nguyen [1,2], Khai-Duy Nguyen [1], Tuan-Kiet Dang [1], Nguyen The Binh [1,2], Cuong Pham-Quoc [2,*], Ngoc-Thinh Tran [2], Cong-Kha Pham [1] and Trong-Thuc Hoang [1]**

[1] Department of Computer and Network Engineering, The University of Electro-Communications (UEC), Tokyo 182-8585, Japan; binh@vlsilab.ee.uec.ac.jp (B.K.-D.-N.); duy@vlsilab.ee.uec.ac.jp (K.-D.N.); tuankiet@vlsilab.ee.uec.ac.jp (T.-K.D.); binh.nguyen288@hcmut.edu.vn (N.T.B.); phamck@uec.ac.jp (C.-K.P.); hoangtt@uec.ac.jp (T.-T.H.)

[2] Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology (HCMUT), 268 Ly Thuong Kiet St., Dist. 10, Ho Chi Minh City 740050, Vietnam; tnthinh@hcmut.edu.vn

* Correspondence: cuongpham@hcmut.edu.vn

**Abstract:** The Trusted Execution Environment (TEE) is designed to establish a safe environment that prevents the execution of unauthenticated programs. The nature of TEE is a continuous verification process with hashing, signing, and verifying. Such a process is called the Chain-of-Trust, derived from the Root-of-Trust (RoT). Typically, the RoT is pre-programmed, hard-coded, or embedded in hardware, which is locally produced and checked before booting. The TEE employs various cryptographic processes throughout the boot process to verify the authenticity of the bootloader. It also validates other sensitive data and applications, such as software connected to the operating system. TEE is a self-contained environment and should not serve as the RoT or handle secure boot operations. Therefore, the issue of implementing hardware for RoT has become a challenge that requires further investigation and advancement. The main objective of this proposal is to introduce a secured RISC-V-based System-on-Chip (SoC) architecture capable of securely booting a TEE using a versatile boot program while maintaining complete isolation from the TEE processors. The suggested design has many cryptographic accelerators essential for the secure boot procedure. Furthermore, a separate 32-bit MicroController Unit (MCU) is concealed from the TEE side. This MCU manages sensitive information, such as the root key, and critical operations like the Zero Stage BootLoader (ZSBL) and key generation program. Once the RoT is integrated into the isolated sub-system, it becomes completely unavailable from the TEE side, even after booting, using any method. Besides providing a secured boot flow, the system is integrated with essential crypto-cores supporting Transport Layer Security (TLS) 1.3. The chip is finally fabricated using the Complementary Metal–Oxide–Semiconductor (CMOS) 180 nm process.

**Keywords:** Trusted Execution Environment; RISC-V; secure-boot; FPGA; VLSI



**Citation:** Kieu-Do-Nguyen, B.; Nguyen, K.-D.; Dang, T.-K.; The Binh, N.; Pham-Quoc, C.; Tran, N.-T.; Pham, C.-K.; Hoang, T.-T. A Trusted Execution Environment RISC-V System-on-Chip Compatible with Transport Layer Security 1.3. *Electronics* **2024**, *13*, 2508. https://doi.org/10.3390/electronics13132508

Academic Editor: Nicola Lusardi

Received: 17 April 2024
Revised: 21 June 2024
Accepted: 22 June 2024
Published: 26 June 2024



## 1. Introduction

In cyber-security research, the isolation between the programs that run on an Operating System (OS) is called the Trusted Execution Environment (TEE) [1]. This is an advanced feature for a secured OS. During a typical boot procedure, the bootloader initializes the OS; subsequently, the actual OS image is loaded into the system memory. Ultimately, the machine successfully starts up and enters the operating system that has been loaded. The entire process depends on the assumption that all components of the boot process function correctly. However, trust in the realm of computers requires more than just that. Consequently, a trust mechanism is essential, an assurance mechanism that allows you to validate the integrity of each stage of the boot sequence [2], ensuring that no compromises have occurred. Subsequently, it is possible to construct a Trusted Execution Environment (TEE) using the trust above. The design idea of TEE is to segregate trusted and untrusted

codes through a divide-and-conquer technique [3]. Typically, isolation is achieved through privilege separation, which effectively establishes a barrier between different programs. Modern TEE models are currently equipped with software- and hardware-based barrier enforcers at various architectural levels. The ultimate objective of TEE is to exclusively permit the execution of authenticated codes while preventing unauthenticated code from running on the trusted side and acquiring any privileges.

For TEE papers, the most popular architectures are Intel Software Guard eXtensions (SGX) [4–7], ARM TrustZone [8,9], and AMD Secure Encrypted Virtualization (SEV) [10]. Over the years, many improvements have been made. For example, Intel SGX has Haven [11], Graphene [12], and Scone [13]; ARM TrustZone has Komodo [14], OP-TEE [15], and Sanctuary [16]; AMD SEV has SEV-ES [17] and SEV-SNP [18]. Those three TEEs (i.e., Intel, ARM, and AMD) are licensed, and any IP modification is strictly prohibited. Recently, with the trending of open-source hardware of RISC-V, many attempts at open-source TEE models were also proposed. Several examples can be listed, including Hex-Five MultiZone [19], Sanctum [20], TIMBER-V [21], CUstomizable and Resilient Enclaves (CURE) [22], and Keystone [23]. Nowadays, almost all smartphones possess a TEE-like characteristic, and numerous organizations, from software to hardware, promote their devices with pre-installed security attributes.

The key mechanism in TEE is the Chain-of-Trust (CoT). It is a link of many cryptographic operations like hash, sign, and verify. At each layer of the OS stack, the upper layer with lower privilege must verify the signature of the lower layer with higher privilege before doing anything [24]. The Root-of-Trust (RoT) is the initial authentication of the CoT system, which serves as its foundation. To ensure security, the Root-of-Trust (RoT) should not be accessible from the Rich Execution Environment (REE) or the Trusted Execution Environment (TEE) processors once the system has been booted. There are many ways to create an RoT. For example, it could be an asymmetrical key pair, a random value, or a pre-signed certificate. In most cryptosystems, the conventional way of implementing RoT is a hard-coded root key in Read-Only Memory (ROM). Creating RoT is extremely important in a cryptosystem; the integrity of the entire TEE depends on the secure boot procedure with such an RoT. The TEE is a self-contained environment that cannot function as the RoT. It is advisable to execute a secure boot process using an RoT and hardware primitives as a standard practice. In most TEE models, the trusted firmware is usually assumed to be properly loaded into the stack before boot because that is actually the job of the hardware, not the software. Therefore, to carry out this initial task, traditionally, TEE models rely on the hardware itself or other Intellectual Properties (IPs); for example, Platform Security Processor (PSP) [25] for AMD SEV, CryptoCell [26] for ARM TrustZone, and Active Management Technology (AMT) [27] for Intel SGX. Regarding RISC-V, since the RISC-V itself is an open-source hardware, many RoT modifications have been proposed directly into the hardware system. Many RISC-V RoT examples are the Rambus CryptoManager [28] and the OpenTitan [29]. The RoT is the designated location for keeping and overseeing the root key and certificates of the device. As per the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) [30], a hardware platform that can securely boot with an RoT must possess the following capabilities: the ability to generate cryptographic keys, wrap and bind keys, seal and unseal keys, incorporate a True Random Number Generator (TRNG), include an integrity measurement feature, and perform key attestation.

The secure boot procedure involving the RoT is one of the challenges faced by the TEE. The remote attestation of the TEE must be performed using the RoT, established through the secure boot procedure. In addition, the demand for Trusted Execution Environments is continuously growing in the Internet of Things (IoT) era. Several new attack vectors have recently been identified that could potentially compromise the secure boot process or expose the root key [31], rendering the entire TEE system susceptible to attacks. It requires a TEE system that can be updated even after manufacturing to deal with the new threats. In this situation, the combination of RISC-V's open-source Instruction Set Architecture

(ISA) and an open-source TEE is a perfect complement. The RISC-V architecture offers a wide range of customizations [32] to create a custom TEE, effectively addressing persistent issues. With the introduction of RISC-V, we can reexamine the hardware architecture to enhance the TEE and reduce the RoT to the silicon level. This can be achieved while still ensuring a safe boot program that is versatile and adjustable. Consequently, in principle, tampering with the silicon RoT entails disrupting the chip manufacturing procedure. Furthermore, leveraging the growing open-source mentality and incorporating cutting-edge security measures, the CoT's capabilities may be further improved. In essence, this proposal's primary contribution lies in introducing a method to isolate the RoT from the TEE processors while allowing the ability to modify the boot sequence.

In summary, the issue of secure boot with an RoT in a TEE system remains a great challenge that needs further investigation. This work is an improvement from our previous work [33] that was published in 2022. The contributions and improvement of this study can be classified into three primary categories, as outlined below:

1. The TEE-HW framework. A high-security computer system must be developed, and an open-source TEE-HW framework must be developed to interface with the Keystone open-source TEE software framework [23]. The suggested TEE-HW framework must address the following requirements. It must be safe, simple to use, adaptable for different security needs, and, most significantly, simple to upgrade with a new defense mechanism. Many architectural features are left optional and can be easily changed by modifying the Makefile system's parameters. The RISC-V open-source community is welcome to reuse the TEE-HW framework's source codes [34]. Future security developers will benefit from such open-source TEE hardware.
2. TEE-HW with cryptographic accelerators. A unique system designed specifically for TEE was created based on the suggested TEE-HW framework. True Random Number Generator (TRNG), Advanced Encryption Standard Galois/Counter Mode (AES_GCM), and Secure Hash Algorithm 3 (SHA-3) are among the several introduced crypto-cores. Besides the required crypto-cores, we also introduced several crypto-cores for the Transport Layer Security 1.3 (TLS 1.3), such as HMAC-SHA2, Digital Signature Algorithm (Ed-DSA or EC-DSA), Rivest–Shamir–Adleman (RSA), and Authenticated Encryption with Associated Data (AEAD). Furthermore, a hidden write-only memory, which is inaccessible to TEE processors, is another feature of the Ed25519 crypto-core. The keys produced by the Ed25519 module will be kept in this write-only memory. We investigated the performance of the suggested TEE hardware with crypto-cores using FPGA and VLSI implementation. We also looked into the TEE boot performance.
3. TEE-HW with isolated RoT. A heterogeneous architecture for RoT-based secure boot flow was suggested by combining an isolated MicroController Unit (MCU) and Linux-bootable TEE processors. While the concealed MCU handles key generation, secure boot, and root key storage, the TEE side typically runs the TEE software stack. After reset, the very first authentication is performed by the hidden MCU. Then, the other crypto-keys are created and stored in memory. Finally, the boot process is transferred to the TEE processors to boot into the Linux kernel. By this setup, all resources are available for the hidden MCU to use, but after boot, all the peripherals inside the hidden MCU are inaccessible by the TEE domain. The secure boot procedure and the remote administration tool (RoT) are no longer within the TEE domain. This makes the secure boot procedure flexible and capable of withstanding potential future threats. The proposed architecture was developed and tested on both FPGA and VLSI on a 180 nm process.

The remaining parts of this paper are structured as follows. Section 2 presents background knowledge, including the Trusted Execution Environment and Keystone. Section 3 presents the crypto-accelerators used in the proposed system. Section 4 reveals the proposed TEE System-on-Chip. Section 5 presents the proposed secured boot flow. Section 6

summarizes the experimental results. The final portion, Section 7, concludes this study and discusses future work.

## 2. Background Knowledge

### *2.1. Trusted Execution Environment*

Generally, remote computing systems are not capable of resolving security issues. For instance, consumers cannot manipulate the tangible elements on their computers. Information can be transferred, and harmful software can be executed remotely on your computer, either from another computer within the same system or from the internet. Hardware manufacturers are striving to provide a reliable mechanism to address these concerns. Therefore, a TEE is introduced. Historically, TEEs have offered three assurances: (1) integrity: ensuring that the code and data remain unaltered and cannot be manipulated, such as by executing unauthorized code within a partition; (2) confidentiality: preventing attackers from gaining knowledge about the runtime content of the application, including secret keys and code control flow; (3) attestation: providing evidence to a remote party that the environment is secure and has not been tampered with.

Trusted Execution Environment aims to provide a state of separation between applications, hence establishing a boundary between different programs. The barrier is commonly implemented using a privilege separation method and enforced by hardware primitives like memory isolation. To separate low-privilege codes (user's apps) from high-privilege codes (OS's services) or vice versa, the previous iteration of TEEs used an essential method of encrypting the codes that need protection and implementing some authentication between the parties involved. Contemporary TEEs are significantly more intricate than those designed for the trusting mechanism. Nevertheless, the central concept remains unchanged. An enclave, which is a standard configuration for a program operating in a TEE, requires a True Random Number Generator (TRNG) to generate keys and various cryptographic functions for tasks such as key creation, hashing, signature, verification, and cipher encryption/decryption. To provide the entire protection of an enclave, current TEEs commonly incorporate a Trusted Firmware (TF) at M-mode. This firmware offers exclusive services that do not depend on the operating system's services. The primary services provided by TF include dedicated memory allocation, cache flushing during enclave context switches, and encryption of messages entering and exiting an enclave. In addition, TF serves as the Trusted Computing Base (TCB) for establishing the trusted domain and preserving the integrity of enclaves' boundaries. Due to the crucial significance of TF, it is imperative to verify the integrity of TF through a secure boot procedure. The authentication of TF is commonly referred to as the RoT in a TEE system.

Every Trusted Execution Environment implementation requires an underlying hardware mechanism as a barrier enforcer. The Physical Memory Protection (PMP) function is the barrier enforcer for the RISC-V architecture. The RISC-V architecture introduces a range of privileged levels, from Machine-mode (M) and extending to User-mode (U). Every authentication is signed by a lower-privileged level and then validated by a higher level. Therefore, the CoT is formed. The initiation of CoT involves the initial verification process during a reset, referred to as the RoT.

RoT can encompass many possibilities, from a random value to a signature signed using an asymmetric key. The RoT must remain unavailable from the TEE processors to provide security once the system has been booted. The keys the RoT created are used to sign, verify, and broadcast to various components of the TEE security architecture. This process ensures that the TEE environment has a Root-of-Trust. The primary objective of TEE is to thwart the execution of unauthorized code on the trustworthy side and prevent it from acquiring any privileges. The TEE employs a combination of cryptographic processes during the boot process to verify the authenticity of the bootloaders. It then validates critical data and applications, such as OS-related programs. Consequently, when booting, we can ensure that only apps that are considered trustworthy will possess valid authentications. Both untrusted codes and infected trusted codes will no longer have valid signatures.

### *2.2. Keystone*

Keystone is a promising open-source project designed explicitly for RISC-V systems. A Keystone enclave can verify its identity, verify the authenticity of software, and ensure the security and confidentiality of remote execution. D. Lee et al. state that it can provide the CoT with secure boot [23], remote attestation, and secure key provisioning. In Keystone, memory isolation at M-mode is achieved through Physical Memory Protection (PMP) and page table isolation. Keystone utilizes RISC-V's PMP feature to implement isolation and prohibit other programs from accessing the enclave memory. Keystone has minimal hardware requirements, including a common RISC-V core, a means to store device keys, and a secure bootloader. Thanks to RISC-V's privilege model and physical memory protection standard, the software can easily manage the remaining tasks. Keystone comprises a collection of software components, rules, and tools that enable the development of TEEs for standard platforms based on the RISC-V architecture. Like SGX-style enclaves, Keystone separates each application into a separate partition during execution. While SGX requires the host to handle all resource management tasks, Keystone enables each enclave to execute user- and supervisor-level code. The system employs a straightforward and adaptable reference monitor, the Security Monitor, which operates below the host operating system to impose security assurances for TEEs. This reference monitor is designed based on the principles of Komodo and Sanctum.

Keystone has multiple memory protection techniques based on specific requirements. As an illustration, the basic setup protects at the software level, cache partitioning can guard against attacks linked to the cache, and on-chip enclave and bus encryption can defend at the hardware level [23]. The Keystone SDK offers essential capabilities necessary for constructing enclave apps. The SDK comprises four components. (1) Host libraries offer an interface for managing enclave applications. (2) Enclave Application libraries provide both essential enclave tools (such as EXIT) and some fundamental libc-style functions (such as malloc and string headers). (3) Edge libraries provide features for managing edge calls to enclave applications and hosts. Edge calls refer to function calls that traverse the boundary between an enclave and its host. (4) Runtime refers to the code at the system level that executes within the enclave. The userland enclave manages the enclave entry point, basic system calls, and all call-related data transfers.

## 3. The Crypto-Accelerators

### *3.1. True Random Number Generator (TRNG)*

Figure 1 shows the block diagram of the proposed True Random Number Generator (TRNG), a part of our proposed system [35]. It is responsible for generating truly random numbers. The generated bits from the TRNG core are stored in the accumulator, which accumulates up to 192 bits. Once the accumulator is full, the bits are shifted into the shift register. When the shift register is filled, the Ready signals are active, and the data are sent out. The final output is stored in a 192-bit register and can be read through Tilelink Peripheral Bus (PBus). The Arbitrator controls the operation of the TRNG core. It determines when and how the generated random numbers are transferred out. When receiving the request from the PBus, the Arbitrator initiates the sampling process. The number of samples is set through 32-bit registers. Per each cycle, the Arbitrator activates the TRNG core to generate random bits and increases the number of samples until it exceeds the set-up samples. The TRNG core passes the non-IID standard test from the NIST. The TRNG will generate the seed for the key generation step of our proposed booting flow in Section 5. In addition, the generated seeds are also necessary for the cryptographic core, as we propose in the following sub-section.

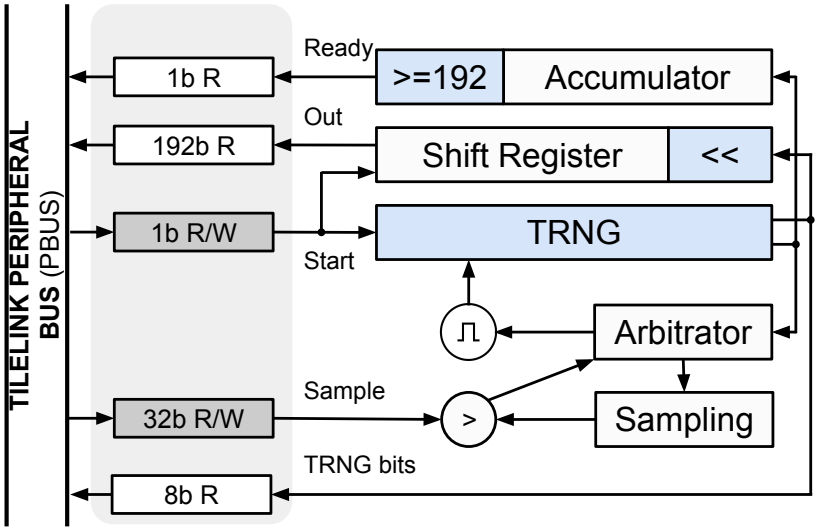


**Figure 1.** The proposed TRNG architecture.

### *3.2. SHA3-512*

The first security accelerator utilized in the TEE boot phase is the SHA3 unit. This accelerator comprises a padding module and a Keccak-1600 calculator [36]. The padding module extracts 64-bit data from the register router and passes it via a 576-bit buffer using a shifter. Once the buffer is filled, the accelerator executes a circular computation. The Constant Counter (see Figure 2) monitors the number of rounds and the consistent non-linearity of the iota phase of the Keccak algorithm. The initial round is computed using the first 64-bit data processed by the padding module. A 1600-bit status register stores the state of each round. After the Padding Module (see Figure 2) processes the final data, the Round Calculation (see Figure 2) executes the last rounds in the status registers. Subsequently, the first 512-bit word can be utilized as the hash result. The results are then read by the processor through PBus. Figure 2 illustrates the proposed SHA3 architecture. The SHA3 unit hashes the private keys used by the Elliptic Curve/Edward Curve Cryptography module, which is used in our proposed boot process. In addition, SHA3 is the selected hash function for TLS 1.3.

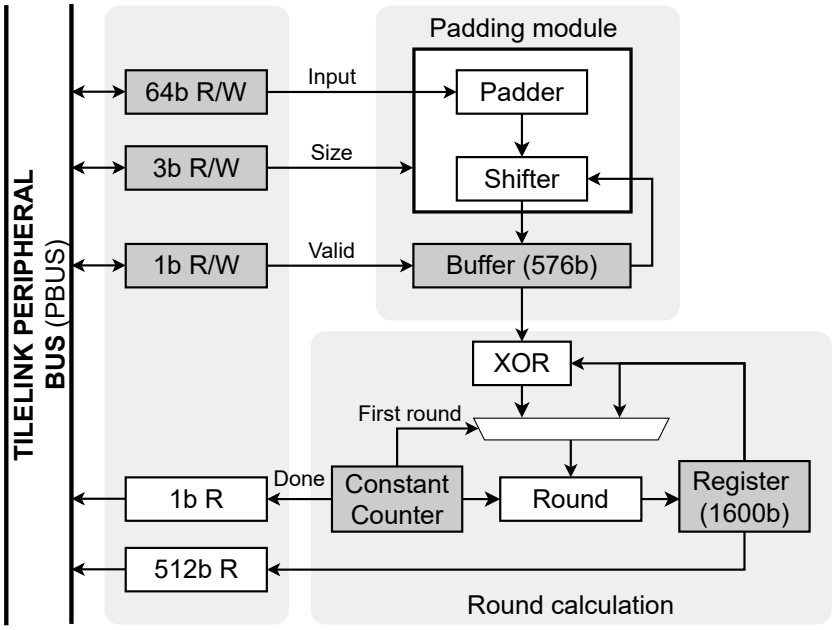


**Figure 2.** The proposed SHA3 architecture.

### *3.3. Advanced Encryption Standard (AES)—Galois/Counter Mode (GCM)*

The second security accelerator implemented is associated with the AES cipher [37]. The AES-GCM accelerator performs encryption or decryption on blocks of either 128 bits or 256 bits, depending on the chosen configuration. It is possible to modify the bit length dynamically. Every datapath executes the Substitution Box (SBox) or Inverted S-Box (InvSBox), shifts rows, mixes columns, and an extra round key. The round key is computed externally for the 128-bit and 256-bit key variants. The AES calculation is executed by a state machine that activates the datapath and indicates data transfer to the output register. Figure 3 illustrates the proposed AES-GCM architecture. AES-GCM is a mandatory encryption of TLS 1.3.

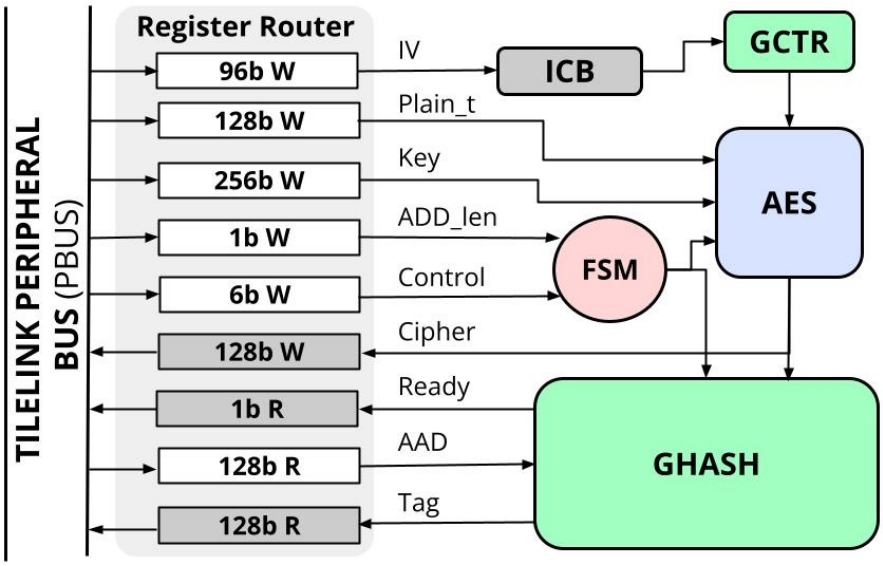


**Figure 3.** The proposed AES-GCM architecture.

### *3.4. Hash-Based Message Authentication Code (HMAC) with Secure Hash Algorithm 2 (SHA2)*

The third security accelerator is HMAC-SHA2 [38]. The integrated HMAC-SHA2 accelerator performs a two-round process to calculate the Hash. In the first round, the inner key is derived from combining the inner pad, a constant string, and the input secret key during the initial step to produce the inner key. Subsequently, the inner key is associated with the input message. This combination is then hashed to generate the digest. In the second round, the digest is combined with the outer key before being hashed (see Figure 4). HMAC-SHA2 is an acronym for Hash-based Message Authentication Code and uses SHA2 as its underlying cryptographic hash function. The implemented SHA2 core can perform four standards, which are SHA2-224, SHA2-256, SHA2-384, and SHA2-512. The input data are expanded during the hashing process, and the input message is divided into chunks. The chunk size for SHA2-224/256 is 512 bits, while for SHA2-384/512 it is 1024 bits. The received data are compressed in the next stage. Lastly, the compressed data are used to compute the new hash value. HMAC-SHA2 is the compulsory authentication scheme of TLS 1.3.

### *3.5. Elliptic Curve (EC) and Edward Curve (Ed) Digital Signature Algorithm (DSA)*

The four security accelerators are the Elliptic Curve and Edward Curve Digital Signature Algorithm (ECDSA/EdDSA). ECDSA and EdDSA generate public and private keys, which will subsequently be employed in signing and verification procedures. ECDSA and EdDSA play pivotal roles in our proposed secured boot scheme. The data are inputted into the memory-mapped Random Access Memory (RAM). The SHA3 hashed private key is read by the processor from the SHA-3 module and written into the ECDSA/EdDSA's RAM. The Processing Elements (PEs) fetch the private key from memory and then multiply it with

the base point of the selected curve. The Finite State Machine (FSM) is pre-programmed in Read-Only Memory (ROM) as the microcode that controls the operations of the PEs. The PEs execute the decoded instructions from ROM using its built-in decoder. The execution units in each Processing Element include adders, subtractors, and multipliers driven by the decoded instructions. Each calculation module has a basic calculator to round the value to the prime number depending on the selected mode, which is essential for the algorithm to execute the operations based on the Edward Curve of the Elliptic Curve. The outcomes of every operation are temporarily stored in RAM, which acts as a register file in this case. The final results are finally written back to RAM through a local bus. The RAM is also used to store constants defined by the selected curve. To enhance the parallelism, the embedded microcode in ROM includes vector-based instructions that effectively control multiple execution units to perform parallel tasks defined in ECDSA and EdDSA specifications. Finally, The Keystone system uses the produced signature to sign the bootloader program, discussed in Section 5. Figure 5 illustrates the proposed ECDSA/EdDSA combinational architecture. ECDSA and EdDSA are the compulsory key exchange schemes required by TLS 1.3.

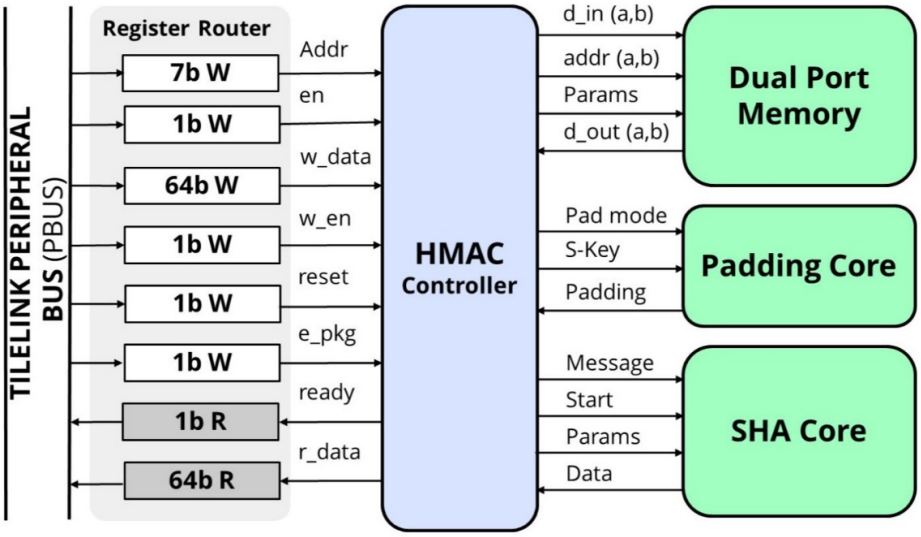


**Figure 4.** The proposed multi-functional HMAC-SHA2 architecture.

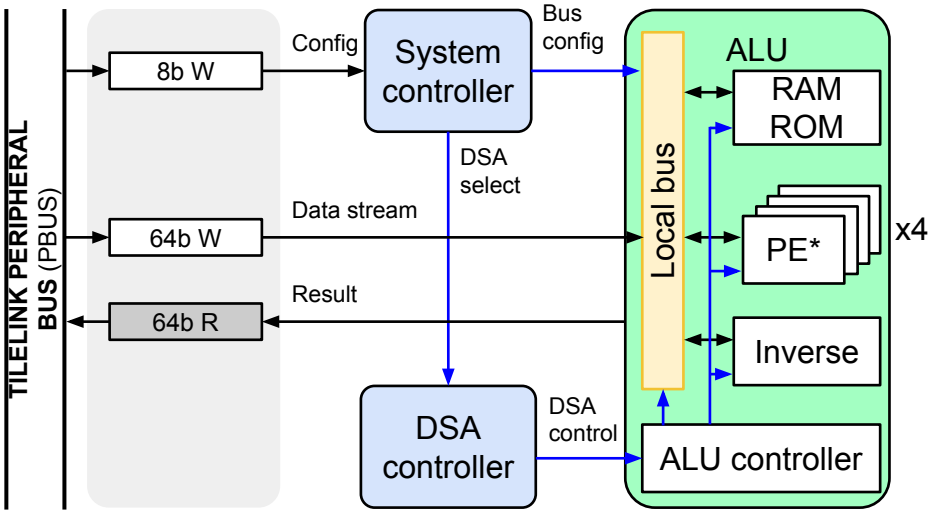


**Figure 5.** The proposed ECDSA/EdDSA combinational architecture. PE*: Processing Element.

### *3.6. Rivest–Shamir–Adleman (RSA)*

The fifth security accelerator is the Rivest–Shamir–Adleman (RSA) module. The proposed architecture is revealed in Figure 6. It uses 1024-bit values to calculate the power model function. To reduce the complexity of the calculation circuit, the 1024-bit register is split into multiple registers with smaller sizes. At the initial stage, the Pre-computation module (getNumBits) (see Figure 6) generates the necessary initial values. The output of these modules is stored in registers. Then, the FSM controls the accumulator ($\pm$) and comparator ($<$) to perform the encrypt/decrypt operations that follow the RSA's specification. The data in the output are 64-bit. Figure 6 illustrates the RSA block diagram. RSA is the compulsory encrypt/decrypt function required by TLS 1.3.

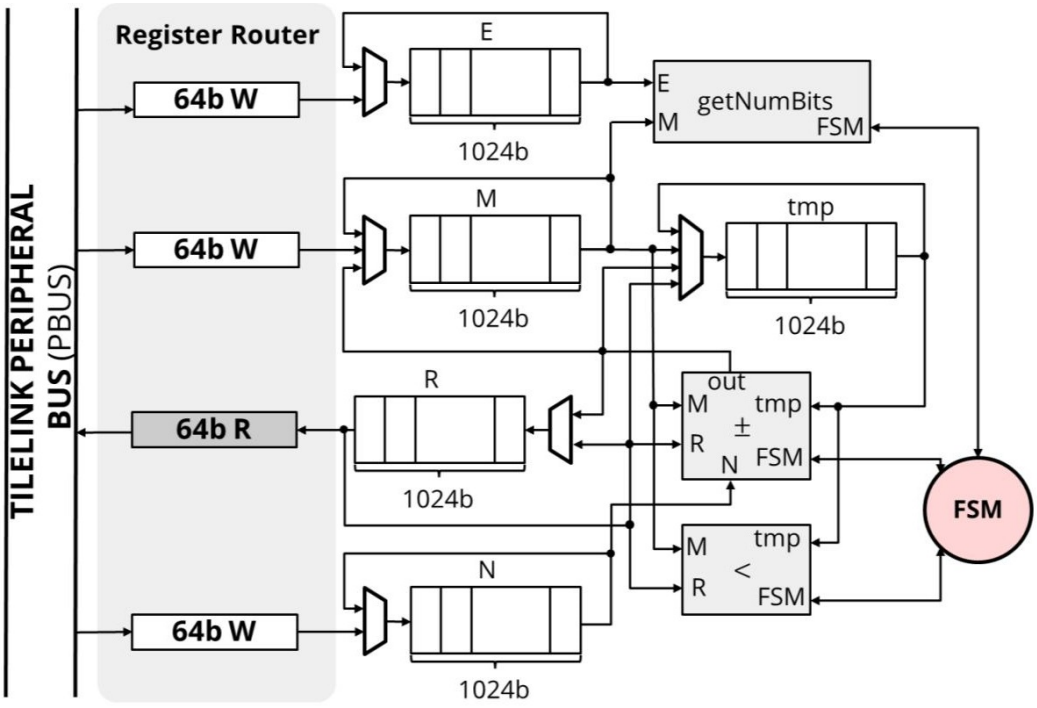


**Figure 6.** The proposed RSA-1024 architecture.

### *3.7. Authenticated Encryption with Associated Data (AEAD)*

The sixth security accelerator is the Authenticated Encryption with Associated Data (AEAD) module. As per the specification, AEAD utilizes a 256-bit key, 96-bit nonce, plaintext of any length, and Additional Authenticated Data (AAD) of any length [39]. Figure 7 depicts the block diagram of the AEAD module, which is connected to the system through the Tilelink Peripheral bus. In the proposed architecture, the key utilized in Poly1305 is derived from ChaCha20. ChaCha20 and Poly1305 algorithms are integrated into a single peripheral to reduce the overhead from data exchange through the shared system bus. Two FSMs are designed to manipulate the operations of ChaCha20 and Poly1305 cores as well as data exchange between these two cores. The ChaCha20 core produces the cipher text based on the input plaintext, key, and nonce (see Figure 7). Meanwhile, the Poly1305 generates the authentication tag (MAC) based on half of the 512-bit generated cipher text of the Chacha20 core and the input Associate Data (AD). The final results, including 512-bit cipher text generated from Chacha20 and 128-bit MAC generated from Poly1305, are then read by the processor through PBus (see Figure 7). AEAD is a recommended encryption scheme by TLS 1.3. It provides better performance than AES-GCM.

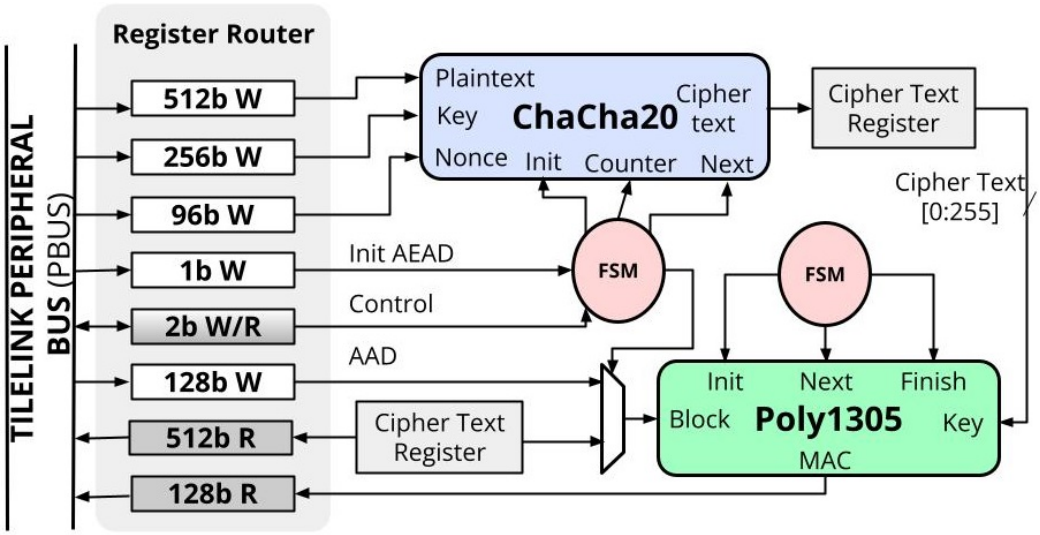


**Figure 7.** The proposed AEAD architecture.

## 4. TEE System-on-Chip

### *4.1. The Isolated Sub-System*

The isolated 32-bit architecture and the standard 64-bit TEE processors are seen side by side in Figure 8. A RISC-V-based RV32IMC IBex [40] core is present in the isolated sub-system. The IBex was selected due to its compact 32-bit core with tamper awareness. The isolated design uses a TileLink bus called IBus as its main bus and a boot ROM. This sub-system also has a separate Core Local INTerrupt (CLINT) and Platform-Level Interrupt Controller (PLIC). For scheduling purposes, internal core-level interrupts are handled by the isolated CLINT. The isolated core can receive commands from the external TEE processors via the PLIC. The PLIC's interrupts are then handled by the IBex core using programs that are kept in its boot ROM.

The TRNG employed in this section is derived from earlier research [35]. As Figure 8 illustrates, PBus connects the TRNG core with the system. The NIST standard demands that the TRNG be in the same environment with the derived keys [41]. Therefore, the TRNG module has two separate PBus connections, one for the data and one for the commands. As a result, the IBex core and the TRNG module have a direct connection, minimizing side-channel attack risk. The TRNG will self-reset after completing its transaction. Thus, TRNG values can be seen as non-independence, non-IID data since the commands originating from the two channels are not regarded equally. The TRNG is proven to pass the non-IID NIST test [35]. Therefore, the two-channel strategy of implementing TRNG did not affect the random quality.

### *4.2. The Isolated TEE System*

Figure 8 displays the suggested design. The architecture shown in Figure 8 also includes a variety of properties, like the number and type of cores, the ISA configuration, and the sizes of the L1 and L2 caches, which are easily reconfigurable based on specific requirements. In addition, each crypto-core, the PCIe connection, and the entire isolated sub-system can be added or removed based on requirements. By default, each core in the dual-core system contains a 16 KB instruction cache and a 16 KB data cache. The Rocket core is ranked first, followed by the BOOM core. The default configuration is RV64GC ISA, 512 KB L2 cache, including the isolated domain and all the peripherals in Figure 8, and excluding the PCIe controller.

The 32-bit isolated MCU is the special feature of this heterogeneous architecture. Upon reset, the isolated MCU boots first; it performs initial authentication and then uses root keys with random integers from TRNG to produce keys. Subsequently, the TEE processors will be activated by the standard TEE boot sequence [23]. The Isolated Bus (IBus) is the

primary bus of the isolated sub-system. It is a master-only TileLink [42] connection with the System Bus (SBus). As a result, all peripherals under the IBus are obscured from the TEE processors. In contrast, the hidden MCU can access every submodule in the SoC. Therefore, the isolated domain is the ideal location for root keys.

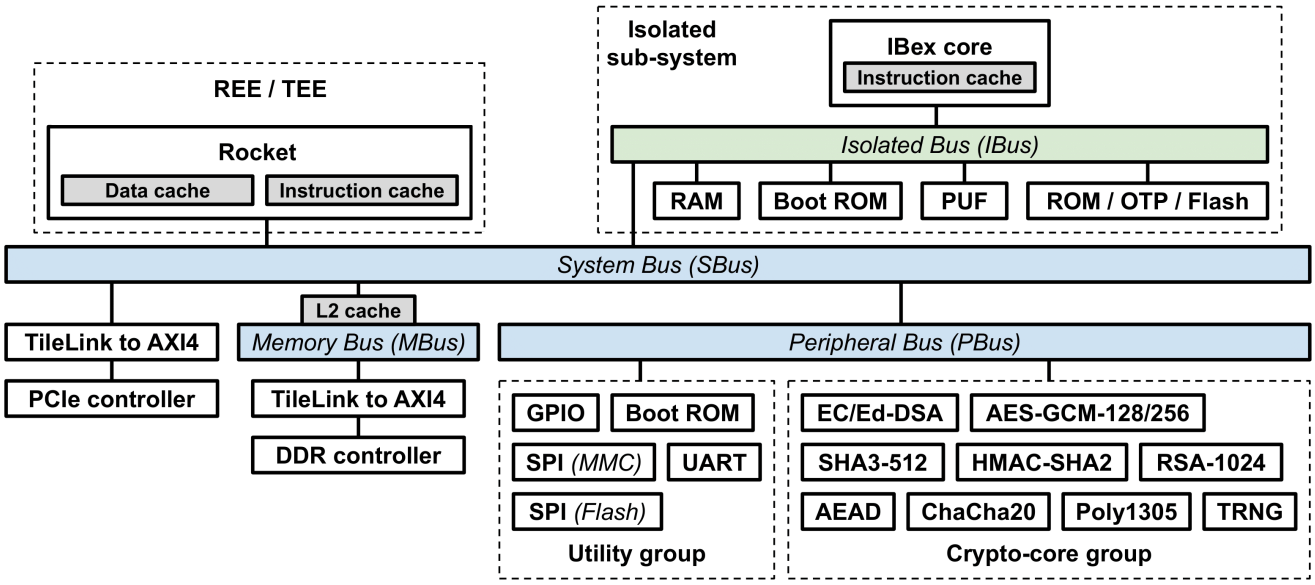


**Figure 8.** The proposed TEE-HW architecture with isolated sub-system.

The L2 cache is integrated with a coherence cache manager. The Peripheral Bus (PBus), as seen in Figure 8, contains a Universal Asynchronous Receiver/Transmitter (UART), several GPIOs, a boot ROM, an SPI for SD card, and an SPI for flashes. For the crypto-core group, several popular cryptographic accelerators are added, including SHA3, DSA, and TRNG. The TEE hardware is also integrated with a DDR controller for booting and running the Operating System (OS) and the software. Finally, to control the external DDR memory, a TileLink-to-AXI4 bridge is used to connect the inside Memory Bus (MBus) with the AXI4 protocol [43] to the outside DDR IP controller. The integrated devices on PBUS, like GPIO, can be exported to the outside for the VLSI implementation. Consequently, the manufactured chip can connect to an FPGA platform and leverage its DDR IP.

## 5. Secured Boot Flow

The Keystone framework [23] is the base for the suggested boot process and key generation. Depending on the Keystone's definitions, two things must be trusted to create a secure boot process. (1) Manufacturer of hardware: Chip makers need to responsible for their products. As a result, we may rely on the silicon manufacturing process to produce trusted hardware, like RoT. (2) Software providers must also adhere to security requirements to safeguard their products. However, there are two reasons why the infrastructure and data transmission environment cannot be used in these two cases. (1) Infrastructure: Many elements could reduce infrastructure security. For instance, security flaws in virtualization software enable hackers to launch direct attacks on other virtual machines from the compromised virtual machine. (2) Data transmission environments: Hackers can intercept data being transmitted via transmission lines, such as the Internet. From these perspectives, we offered a safe boot flow based on the isolated TEE system.

Initially, the RoT utilizes a secure chip with a trusted boot ROM to generate a hash of the software binary, creating $H_S$. The programs that require hashing are the sensitive programs, such as OS-related applications and those that require a specific privilege after booting. Every software possesses its uniquely produced $H_S$. Once the $H_S$ is made, the $SK_D$ and $H_S$ are utilized to build the software pair keys, which consist of the secret key $SK_S$ and the public key $PK_S$, using a Key Derivation Function (KDF), as depicted in the figure.

Once the $SK_S$ and $PK_S$ have been generated, the $SK_D$ is employed to sign and validate the $PK_S$ together with its $H_S$, creating a software certificate. The $Cert_S$ can now be utilized to authenticate the software's integrity, as it is securely linked to $H_S$ and endorsed by the device. We can generate an attestation report that traces back to the original manufacturer by utilizing a series of certificates. Once all the required certificates have been generated, the machine can boot into the operating system space. Due to the lack of trustworthiness of the boot image $S$, it is necessary to remove all sensitive data beforehand, such as the stack and $SK_D$.

Figure 9a shows the secure boot procedure and the Keystone boot flow performed by the heterogeneous architecture of TEE and hidden processors. The key idea is that the chip manufacturer will act as the root Certificate Authority (CA). Therefore, the root CA's public key $P_M$ is widely recognized, and the root CA's certificate $M_{Cert.}$ is a self-signed certificate. Each manufacturer can have multiple key pairs, but each key pair is unique for its manufacturer. Since the key pairs of $S_M$ and $P_M$ are generated offline, it is advisable to utilize high-bit RSA keys with an extended validity period of several years.

To improve the security level, the $S_R$ and $P_R$ root keys should be elliptic curve keys. These keys are generated by the manufacturer during the offline design phase. As mentioned earlier, the $R_{Cert.}$ root certificate is a self-signed certificate using the secret key $S_M$ of the manufacturer. Furthermore, the root secret key $S_R$ is not saved anywhere, but the root public key $P_R$ is stored in the boot ROM of the isolated domain. The purpose $P_R$ is for the first authentication in the ZSBL. The isolated boot ROM also holds the very first boot loader called the secure BootLoader (sBL). As shown in Figure 9a, the very first task of the hidden processor is to verify the sBL content using the $S_M$. The hidden processor of IBex is also the core that runs the sBL content, which involves verifying and loading the ZSBL using the provided $P_R$.

The subsequent stage generates the EC key pair, $S_D$ and $P_D$, for the device/chip. As depicted in Figure 9a, the manufacturer also produces them offline. The confidential key $S_D$ of the device is securely stored in a separate Read-Only Memory (ROM). In contrast, $P_D$, the device public key, is stored in a publicly accessible location. The ZSBL is located in the same place as the $P_D$. Its job is to verify the signature signed by $S_R$, the root secret key. Since the isolated processor's initial action involves verifying and loading the ZSBL, this approach enables the manufacturer to securely update it, even if it is kept publicly, such as in an off-chip nonvolatile memory.

Once confirmed and loaded, ZSBL utilizes the True Random Number Generator as a seed for the EC-genkey algorithm to generate a pair of subsequent keys, namely $S_K$ and $P_K$, referred to as Keystone keys. The keys are stored in a publicly accessible Random Access Memory (RAM) on the Trusted Execution Environment's side. Next, the secret key $S_D$ of the device is utilized to sign the public key $P_K$ of the Keystone, resulting in the creation of the Keystone certificate $K_{Cert.}$, as depicted in Figure 9a. Subsequently, the FSBL's content is transferred from the SD card to the main memory of the TEE domain, where the $S_D$ is used to hash and sign its content. In the next step, both the FSBL and Keystone key pair are stored in the main memory and prepared for execution by the TEE processors. Ultimately, the isolated core activates the TEE processors to continue the conventional TEE boot flow.

Figure 8 depicts the connection between the IBus and the SBus, which uses the master-only TileLink protocol. As a result, all the resources below the SBus can be reached through the IBus, but not vice versa. Therefore, the TEE processors cannot access information within the hidden MCU due to this master-only bus. The Read-Only Memories (ROMs) and Random Access Memories (RAMs) located within the hidden MCU are well-suited for storing keys and carrying out the secured Bootloader (sBL) and Zero Stage Bootloader (ZSBL) operations. During the boot process, the isolated sub-system will be the first to boot to establish the RoT. Figure 9b illustrates the program execution process within a controlled environment. This application will execute once the system has been restored to the reset state.

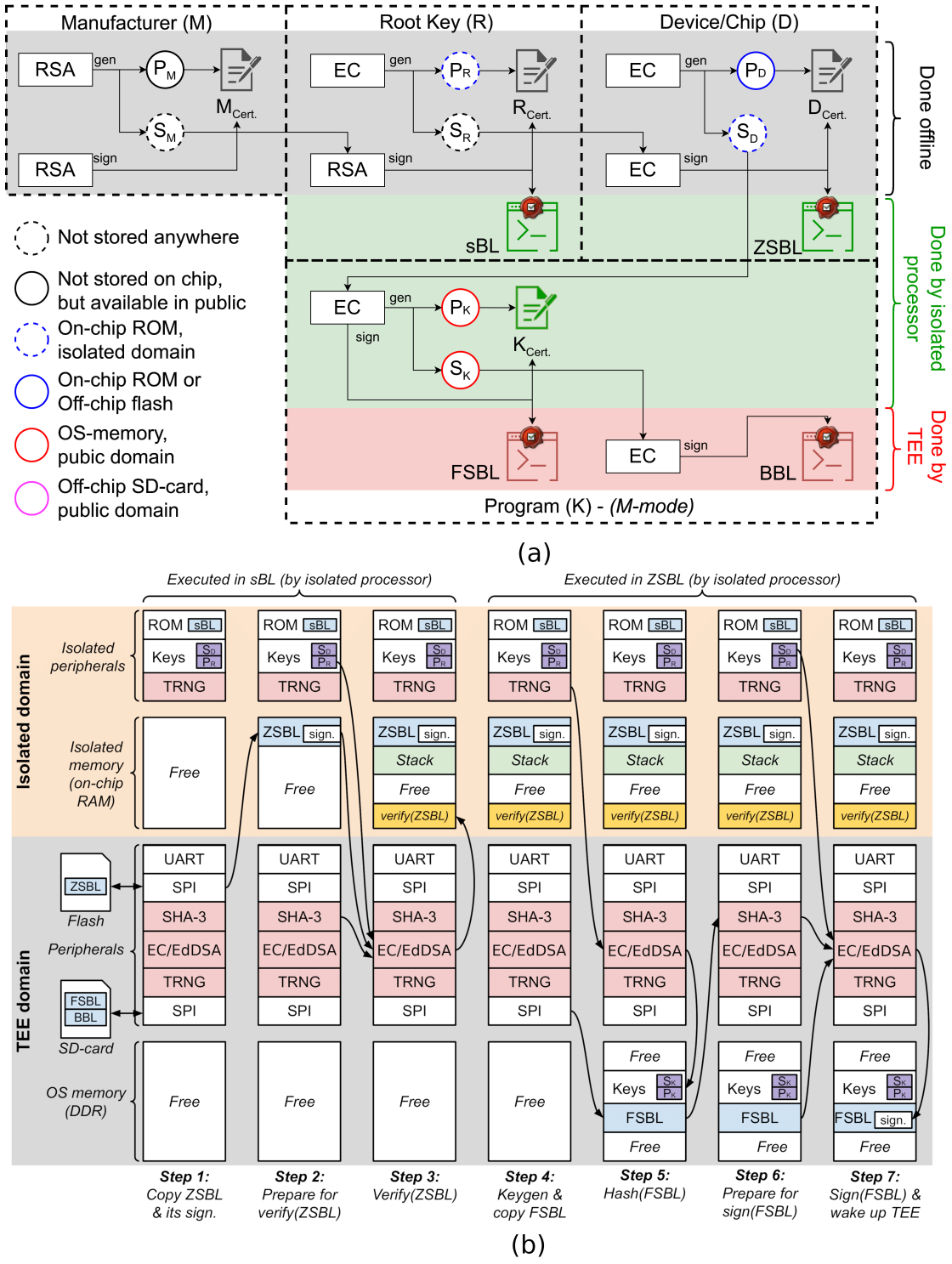


**Figure 9.** (**a**) Key management in the secure boot procedure. The recommended storages for sBl, ZSBL, FSBL, and BBL are the isolated ROM, off-chip flash, SD card, and SD card/hard drive, respectively [33]. (**b**) Boot flow in the isolated environment (green part of **a**).

Upon reset, the TEE processors, a key component of our system, will enter a state of waiting for interrupt. With the support of the crypto-accelerators, the isolated sub-system retrieves the root/device keys from the ROM and combines them with TRNG to form a seed for Ed25519, resulting in the generation of the Keystone key pair. Once the $S_K$ and $P_K$ are created, the $S_K$ will be stored in a write-only memory. This memory, designed explicitly

for the Ed25519 crypto-core, can only be accessed by crypto-cores for their operations. TEE processors and even the IBex core cannot read from this memory. Cryptographic cores can utilize this non-writable memory to compute the signature of a sequence of bits. In this scenario, the OS Bootloader (*S*) undergoes a process of hashing and is then internally signed using the previously saved private key of the Curve25519 function.

Because of the absolute separation between the two domains, it is impossible for any external program on the TEE processor to manipulate the operations of the isolated domain. This architectural feature is a strong defense against unauthorized access. The main possible threat from the TEE side is the interrupt exploitation requesting the authentication from the hidden sub-system. However, since the IBex's behavior solely depends on the program in its isolated boot ROM, we can reprogram the IBex core to cope with new threats. This flexibility in our system's design is another layer of defense against potential threats.

## 6. Experimental Results

### *6.1. Experimental Setups*

The proposed Trusted Execution Environment hardware system supports Rocket and Ibex with Instruction Set Architecture (ISA) settings of RV32GC and RV32IMAC. The proposed TEE-HW SoC in Figure 8 was implemented in both FPGA (Xilinx Virtex-7 XC7VX485T) and VLSI (CMOS 180 nm technology). A single-core RV32GC Rocketchip was used as the TEE processor. It has 16 KB of instruction cache and 16 KB of data cache. For the hidden sub-system side, the IBex core with 4 KB of instruction cache was used. Compared to Figure 8, the PICe module is excluded, and the utility and crypto-core groups are included.

### *6.2. Resource and Power Consumption*

The proposed system was implemented and tested on the Virtex-7 FPGA with the chip series of XC7VX485T; the results are given in Table 1. As seen from the table, the EdDSA/ECSA module occupied almost half of the design with 42.61% LUTs and 12.84% registers. For the whole design, 31.9% of the FPGA resources were spent. The isolated sub-system costs only 5.08% of LUTs, nearly half compared to the Rocket core. Table 1 provides the resource consumption for variable crypto-cores. The relation between the used Lookup Tables (LUTs) and Registers in Table 1 is illustrated in Figure 10.

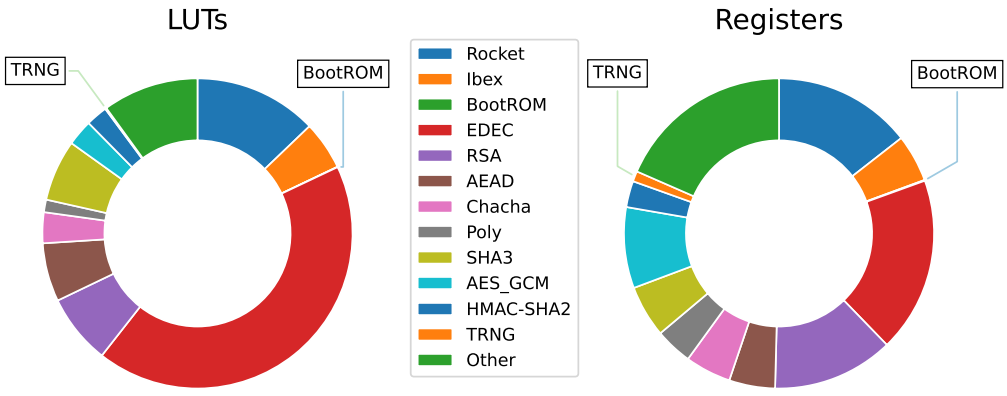


**Figure 10.** FPGA resource consumption.

For ASIC implementation, the proposed SoC was synthesized in the conventional bulk CMOS 180 nm process. Figure 11 illustrates the layout and micro-graph of the fabricated chip. The results of the system with 100-MHz constraints are given in Table 2. The sizes of the chip and the submodules are revealed in Tables 2 and 3. According to the comparison table, nearly a third of the area was dedicated to the Rocket-tile at 34.59%, while the power consumption is just 13.82%. The Ed-DSA combined with the EC-DSA, the EDEC module, consumes the most power at 42.63% while costing 24.68% of the area. The whole hidden

sub-system, the IBex-tile, is quite small, with only 5.00% area and 2.24% power. The relation of sizes versus powers of different components in the TEE SoC is revealed in Figure 12 following the statistics in Table 2.

**Table 1.** Proposed 32-bit TEE SoC performances on Virtex-7 FPGA.

| Instance | LUTs | | Registers | | BRAM | Size (KB) | DSP Blocks |
|---|---|---|---|---|---|---|---|
| Total system | 97,040 | 100.00% | 52,099 | 100.00% | 28 | 619 | 16 |
| Rocket | 12,465 | 12.84% | 7530 | 14.45% | 12 | 544 | 2 |
| core | 3478 | 3.58% | 1521 | 2.92% | 0 | 0 | 2 |
| dcache | 2107 | 2.17% | 3716 | 3.77% | 2 | 16 | 0 |
| icache | 5982 | 6.16% | 3716 | 7.13% | 2 | 16 | 0 |
| Ibex [1] | 4929 | 5.08% | 2575 | 4.94% | 2 | 8 | 0 |
| BootROM | 38 | 0.04% | 43 | 0.08% | 4 | 32 | 0 |
| EDEC | 41,353 | 42.61% | 9524 | 18.28% | 8 | 2 | 0 |
| RSA | 7087 | 7.30% | 6589 | 12.65% | 0 | 0 | 0 |
| AEAD | 5925 | 6.12% | 2497 | 4.79% | 0 | 0 | 7 |
| Chacha | 3118 | 3.21% | 2497 | 4.79% | 0 | 0 | 0 |
| Poly | 1268 | 1.31% | 2023 | 3.88% | 0 | 0 | 7 |
| SHA3 | 6200 | 6.39% | 2820 | 5.41% | 0 | 0 | 0 |
| AES_GCM | 2635 | 2.71% | 4400 | 8.45% | 0 | 0 | 0 |
| HMAC-SHA2 | 2178 | 2.24% | 1425 | 2.74% | 2 | 1 | 0 |
| TRNG | 136 | 0.14% | 563 | 1.08% | 0 | 0 | 0 |
| Other * | 9708 | 10.00% | 9613 | 18.45% | 0 | 0 | 0 |

[1] Including the isolated sub-system. * Bus system, debug module, peripherals, interrupt.

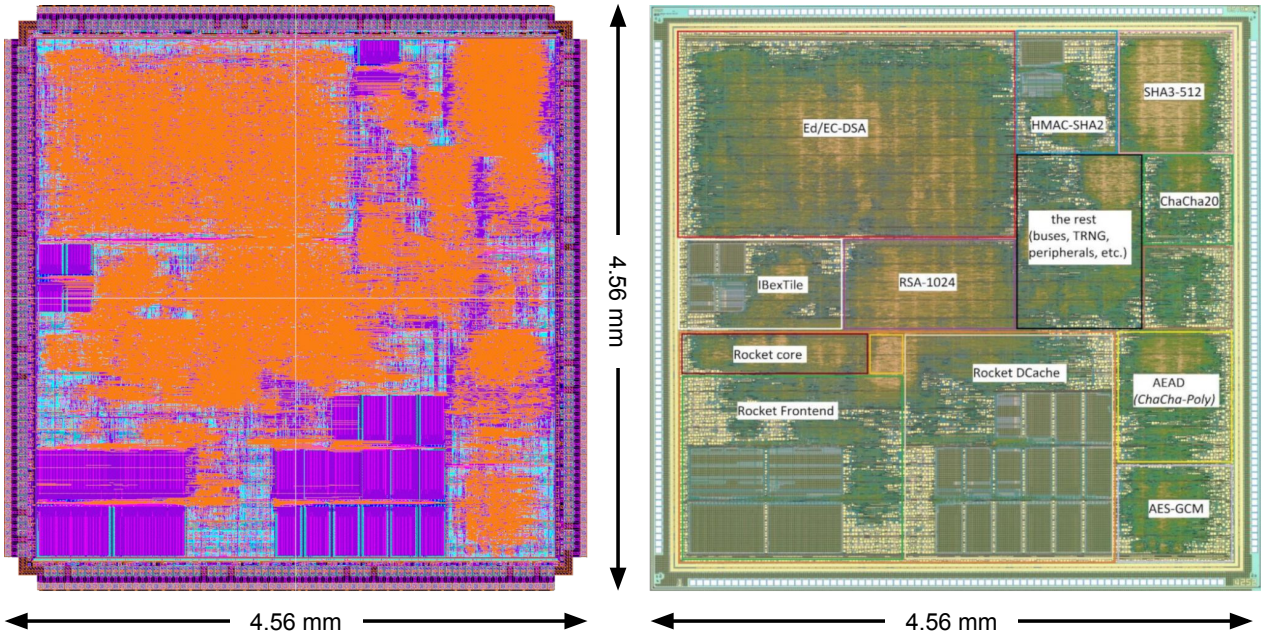


**Figure 11.** ASIC layout and micro-graph.

**Table 2.** Proposed TEE SoC in CMOS 180 nm synthesis result.

| | Gate Eqiv. | Area | | Power | | | |
|---|---|---|---|---|---|---|---|
| | (NAND2) | $\mu m^2$ | % | Leakage (nW) | Dynamic (mW) | Total (mW) | % |
| Total | 460,195 | 14,744,115 | 100.00 | 5487 | 3075 | 3075 | 100 |
| Rocket | 75,030 | 5,100,826 | 34.59 | 1213 | 425 | 425 | 13.82 |
| core | 15,337 | 372,392 | 2.53 | 177 | 182 | 182 | 5.92 |
| dcache | 25,398 | 2,509,375 | 17.02 | 456 | 154 | 154 | 5.01 |
| icache | 32,127 | 2,169,710 | 14.72 | 555 | 77 | 77 | 2.50 |
| IBex [1] | 17,681 | 737,478 | 5.00 | 201 | 69 | 69 | 2.24 |
| BootROM | 4272 | 70,672 | 0.48 | 21 | 11 | 11 | 0.36 |
| ECED | 166,720 | 3,638,115 | 24.68 | 1664 | 1311 | 1311 | 42.63 |
| RSA | 35,754 | 827,563 | 5.61 | 385 | 226 | 226 | 7.35 |
| AEAD | 30,345 | 783,675 | 5.32 | 349 | 223 | 223 | 7.25 |
| Chacha | 16,723 | 402,309 | 2.73 | 178 | 85 | 85 | 2.76 |
| Poly | 10,966 | 308,602 | 2.09 | 136 | 89 | 89 | 2.89 |
| SHA3 | 26,873 | 669,773 | 4.54 | 292 | 156 | 156 | 5.07 |
| AES_GCM | 20,753 | 532,594 | 3.61 | 266 | 80 | 80 | 2.60 |
| HMAC-SHA2 | 13,155 | 529,278 | 3.58 | 176 | 92 | 92 | 2.99 |
| TRNG | 268 | 3983 | 0.03 | 1 | 0.15 | 0.15 | 0.01 |
| Other * | 41,655 | 1,139,247 | 7.74 | 605 | 308 | 308 | 10.03 |

[1] Including the isolated sub-system. * Bus system, debug module, peripherals, interrupt.

**Table 3.** Summary of TEE-HW with isolated architecture chip features.

| Process | | CMOS 180 nm |
|---|---|---|
| Cores | | Rocket |
| ISA | | RV32GC |
| Caches | Instruction<br>Data<br>L2 | 16-KB (Rocket) + 4-KB (Ibex)<br>16-KB (Rocket) + 4-KB (Ibex)<br>512-KB |
| Area | Die | $5.0 \times 5.0$-$mm^2$ |
| | Core | $4560.52 \times 4561.16$-$\mu m^2$<br>$=20.79$-$mm^2$<br>$\approx$1,535,406-NAND2 |
| | Cell | 466,882 |
| | MOSFET | 7,982,582 |
| $V_{DD}$ | I/O<br>Core | 1.8-V<br>1.0-V to 2.0-V |
| Peak performance | | at 2.0-V $V_{DD}$<br>$F_{Max}$ = 30-MHz<br>$P_{Active}$ = 7.6-mW/MHz |

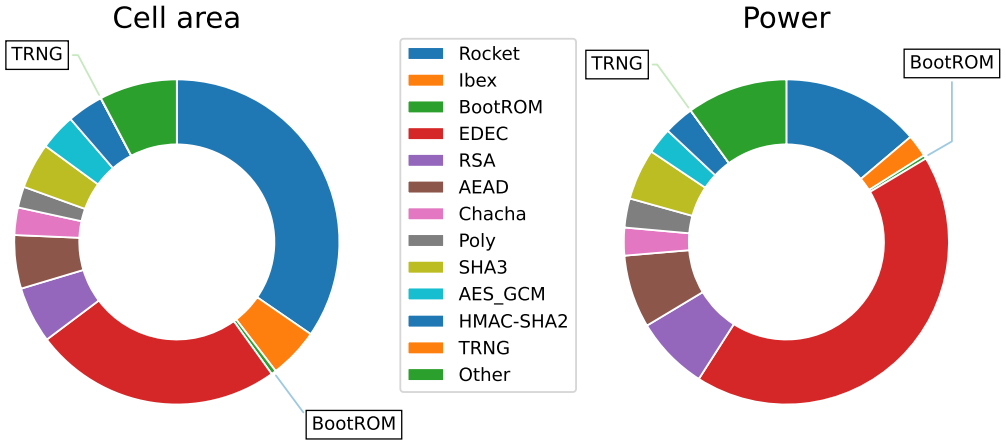


**Figure 12.** Cell area and power.

### *6.3. Performance Analysis*

Besides an FPGA-based implementation, CMOS 180 nm chips were made for the demonstration. For better stability, the critical peripherals, such as Secure Digital card (SD card), Universal Asynchronous Receiver/Transmitter (UART), and Flash rely on Peripheral MODdule (PMOD) headers, were acquired from Digilent and attached with small circuits outside. Similar to the previous PCBs, this PCB also can choose a power supply and clock source. The power and clock can be provided by the FPGA or external sources via jumpers. Figure 13 shows the working PCB mounting on the TR5 FPGA board to use the FPGA's Dual In-line Memory Module (DIMM) Random Access Memory (RAM). In this way, we ensure that the peripherals, especially DIMM-RAM, work properly.

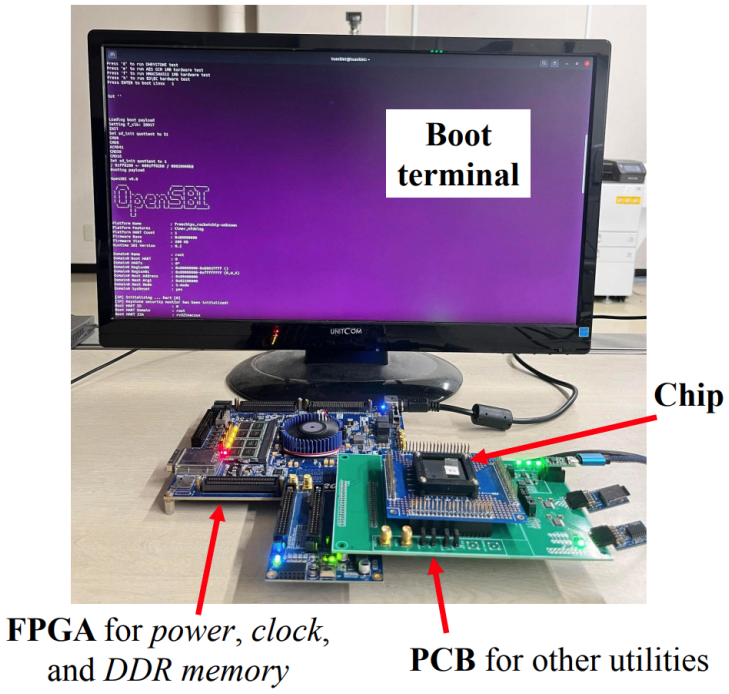


**Figure 13.** The TEE-HW with isolated architecture PCB mounts on the TR5 FPGA board.

In CMOS 180 nm technology, the default $V_{TH}$ is about 1.0 V and the recommended operating $V_{DD}$ is 1.8 V. Therefore, the CMOS 180 nm chip measurement was carried out with the $V_{DD}$ range of 1.0 V to 2.0 V. The system is measured and works at a voltage of 30 MHz from a voltage higher than 1.2 V. However, it can work at the voltage from 1.0 V to 1.2 V for frequencies lower than 10 MHz. Figure 14 shows the changes in power and energy with different $V_{DD}$ for the 32-bit 5.0 × 5.0-mm$^2$ version. The statistic is collected

for three cases, including 30 MHz (which is the maximum frequency overall), 10 MHz (which is the maximum frequency at which the system can work in all ranges of voltage), and 1 MHz (which is the minimum voltage at which the system could work). Because there is a huge gap among active power $P_{active}$, which is the power when the system works, idle power $P_{idle}$, which is the power when the system does not work, and sleep power $P_{sleep}$, which is the power when the input clock is cut off, we normalize the power by the function $power_{normalized} = 3 * \log_{10}(power)$. While the sleep power $P_{sleep}$ is almost identical for different scenarios, the active power $P_{active}$ increases with the $V_{DD}$ and the frequency. Despite having the highest frequency after place and route, at 71 MHz, the fabricated chip can only work stably at 30 MHz due to the limitations of bonding and packaging techniques. Figure 14 also shows the active energy $E_{active}$ and the idle energy $E_{idle}$. Despite the power being small and having a low frequency, the increase in execution time causes a reduction in power efficiency. The system achieves the best power efficiency, which is 7.6 W/MHz, when working at 30 MHz. Table 3 summarizes the features of the TEE SoC on CMOS 180 nm. Although the system's memory is identical between FPGA and ASIC deployment, the ROMs and small-sized BRAMs are converted to registers instead of SRAMs to reduce the delay. Therefore, the total size of SRAMs on ASIC is smaller than that of BRAMs on FPGA.

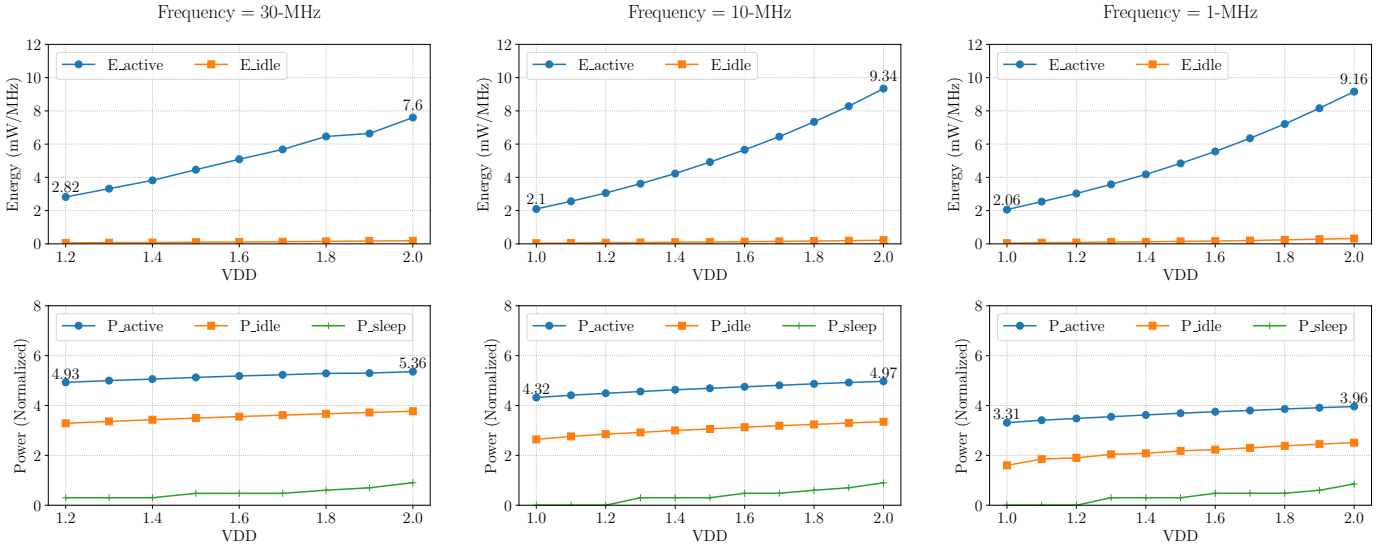


**Figure 14.** ASIC power and energy consumption.

### *6.4. Security Analysis*

The goals of the proposed TEE-HW are (i) accelerating the boot process by using crypto-cores and (ii) isolating the boot program with RoT from the TEE processors. The used TEE model in the implementation is Keystone [23]; thus, the proposed design inherits all of Keystone's advantages and disadvantages.

To summarize Keystone, the threat model of Keystone considers a strong software adversary that can compromise all of the software stack and a strong physical adversary that can corrupt peripherals and memory communications. A malicious enclave is also considered in the Keystone model. In Supervisor-mode (S-mode), the Eyrie runtime provides the Operating System (OS) equivalent services and ensures the validity of address mappings, thus preventing mapping attacks [44]. Furthermore, thanks to the runtime, the enclaves do not have to rely on the OS for critical functions; hence, they can defend against controlled SCAs that exploit the sharing states across domains, like interrupt handlers and table paging. In Machine-mode (M-mode), the Physical Memory Protection feature defines the memory access rights. Therefore, the PMP will deny any direct attempt to read the enclave's data, protecting against the software adversary [23]. In Keystone, the Secure Monitor (SM) performs a clean context switch by flushing enclave states. Together

with cache partitioning, the cache-based SCAs are prevented. Moreover, an enclave can be encrypted, and its page table can be self-managed; thus, any subtle attacks like controlled SCAs are impossible. Finally, Keystone also offers plugins to strengthen the TEE, such as memory encryption, enclave dynamic resizing, edge call service, and syscall services [23]. In Keystone and our implementation, the speculative attacks, timing SCAs, and SCAs that exploit hardware flaws in the off-chip components are considered out of scope. Finally, although possible, enclave-to-peripheral binding is not recommended in the current implementation. Introducing a peripheral driver into the runtime is not a two-way binding process, thus allowing Direct Memory Access (DMA)-capable peripheral attacks.

Regarding the boot flow, most TEE models, Keystone included, generally consider the RoT-based secure boot flow to be out of scope. That makes sense because, by definition, TEE is an isolated environment, not the RoT, and it should not include RoT. It is recommended that the RoT-based secure boot process be run by hardware primitives rather than the TEE processors themselves. Typically, propriety TEE models use third-party IPs or some extra hardware mechanism to deliver the secure boot. In Keystone, the trusted domain operates based on the assumption that the hardware signed the SM during boot. Therefore, RoT hardware is needed to deliver that secure boot process. By introducing a secure boot mechanism with silicon-level RoT to complete the CoT, the device's integrity is guaranteed.

With all of the cryptographic keys interlocked with each other, a direct attack on the key chain is impossible. For example, the public root key $P_R$ and the secret device key $S_D$ are stored in the hidden ROM inside the isolated domain. $S_R$, the secret root key, is not stored in the SoC or anywhere. In the public domain, only the public device key $P_D$ is available after boot and for verification. Additionally, due to the isolation dictated by the bus architecture, even if a malicious enclave could hack the TEE side, it cannot retrieve any data in the hidden ROMs by any means. From the software perspective, exploiting the interrupt channel for attestation is the only attack surface left. But, as mentioned earlier, the IBex core's behavior is solely dictated by its program inside the isolated domain. Thus, if such an attack threat exists, the IBex program can be updated anytime to adapt to the new attack vector. To conclude, the proposed secure boot scheme can still safeguard its secrets even if the TEE processors were compromised. Our fabricated VLSI chip successfully boots with the proposed boot flow (see Figure 15).

### *6.5. Comparison and Discussion*

Table 4 shows the Drystone test results between cores. Due to the difference in the ISAs, for a fair comparison, the Drystone test was repeated by the same IBex ISA of RV32IMC. For each test, the Dhrystone program was run 500 times, and the average value was recorded. As seen from the table, the Rocket achieved a good result of 1.573∼1.713-DMIPS/MHz, while the 0.434-DMIPS/MHz result of IBex is a mid-range processor [45]. Specifically, the Rocket's DMIPS/MHz results were about 3.62× to 3.95× compared to IBex's. Although the IBex core is much slower than Rocket's, considering it still can use crypto-cores to accelerate its boot program, the boot speed when swapping a Rocket for an IBex is not much of a change.

**Table 4.** Dhrystone test comparison between IBex and Rocket cores.

| Core | ISA | Dhrystone/s | DMIPS/MHz | Changes |
|---|---|---|---|---|
| Rocket | RV64GC | 150,511 | 1.713 | 3.95× |
| | RV32IMC | 138,197 | 1.573 | 3.62× |
| IBex | RV32IMC | 38,165 | 0.434 | 1.00× |


**Figure 15.** Boot on chip.

Table 5 compares the security and flexibility features. In ITUS [46,47], they try to solve the secure boot in TEE by a pure hardware approach. The new hardware modules, the Code Authentication Unit (CAU) and Key Management Unit (KMU), have been introduced. The KMU handles the key generation and key distribution by utilizing a PUF and a TRNG. For the CAU, an EC-DSA and an SHA-3 were used for authentication. Because its approach is solely hardware, it lacks the flexibility to adapt to new threats. In contrast, our suggested isolated sub-system is flexible and can be programmed for any cryptographic function. Compared to the works in [46,47], our proposed system has enough crypto-cores to provide a secure boot required by the TLS-1.3 standard.

**Table 5.** Comparison in terms of security and flexibility with recent security-driven implementations; ●, ◐, and ○, respectively, rank the performance from best to worst.

| | CURE [22] | HECTOR-V [48] | WorldGuard [49] | ITUS [46,47] | This work |
|---|---|---|---|---|---|
| Open-source | ○ | ○ | ◐ | ○ | ◐ |
| Secure boot | ◐ | ● | ◐ | ● | ● |
| Flexible boot | ● | ● | ● | ○ | ● |
| TEE isolation | ○ | ○ | ○ | ● | ● |
| Exclusive TEE processor | ◐ | ● | ◐ | ○ | ○ |
| Exclusive secure storage | ○ | ● | ○ | ● | ● |
| Secure I/O paths | ● | ● | ◐ | ○ | ○ |
| Crypto. accel. | ○ | ○ | ◐ | ● | ● |
| SCA resilience | ● | ● | ◐ | ○ | ○ |
| Hardware cost | ● | ◐ | ● | ○ | ◐ |
| High expressiveness | ◐ | ● | ◐ | ○ | ◐ |
| Low porting efforts | ○ | ○ | ◐ | ● | ● |

WorldGuard [49] enhances the security level of TEEs by implementing various IDs across the entire system; this improves the isolation between various OS stacks. However, because their goal is not the secure boot flow but to strengthen the existing TEE models, they use the conventional boot flow for the secure boot process. Specifically, they use the bootloader with hard-coded root keys pre-stored in the ROM. This bootloader is executed first to verify and load the secured channel into the main memory; that means both the boot program and the RoT are still in the TEE domain. Therefore, the WorldGuard approach is still vulnerable to conventional software-based side-channel attacks.

In HECTOR-V [48], the design comes from two novel ideas: the heterogeneous architecture for separating REE and TEE domains and the security-hardened TEE processor for SCA resilience. In HECTOR-V, the TEE processor is the one to execute the secure boot process. That means its secure boot program can be updated in the same way as our approach. However, because the secure boot program is still accessible from the TEE processor, and the REE and TEE share the same processor, there is still a risk of exposing the RoT to the public side, even though the secure storage element was introduced. In contrast, our method completely moves the RoT and its secure boot program from the TEE's eyes, thus eliminating the potential threats from the malicious TEE's enclaves.

CURE [22] is a new model that uses new hardware primitives to raise the security strength and fine-tune various TEE applications. Their implementation can support many types of enclaves simultaneously without affecting the isolation between them. In order to do that, many hardware modifications are introduced, from registers in the core and shared caches partitioning to the bus controller. Although the CURE implementation has achieved solid work for TEEs, it still assumes that the RoT was carried out during the reset. Therefore, regarding the RoT-based secure boot flow, CURE did not provide a solution other than the conventional method of hard-coded keys in ROM.

## 7. Conclusions

This work proposes a Trusted Execution Environment HardWare (TEE-HW) framework that is easy to use, flexible for various needs, and easy to update in the future. The framework offers not only a secured boot process but also sufficient crypto-accelerators, which are required by TLS 1.3. Based on the framework, a completed TEE-HW computer system was developed and tested. The proposed TEE-HW architecture contains several cryptographic accelerators to enhance boot performances and increase security. Finally, a heterogeneous architecture with an isolated sub-system was developed. The hidden Micro-Controller Unit (MCU) in the isolated architecture provides not only the secure RoT implementation but also the ability to adapt to the future changes of the boot sequence. The

architecture contains several crypto-cores, such as AES_GCM, SHA3, and TRNG. Besides the essential cores for the boot process, the crypto-accelerators, such as HMAC-SHA2, Ed/EDDSA, RSA, and AEAD, allow the system to perform different secured protocols. The crypto-cores have been proven to be efficient not only for performance but also for security strength. The proposed TEE-HW SoC was tested on Field-Programmable Gate Array (FPGA) and then realized in a Very Large-Scale Integrated circuit (VLSI). Fabrication is performed with the CMOS 180 nm process, and the measurements are delivered.

There are some limitations that we are working to improve. Firstly, despite the provided framework supporting a secured boot process, the protection against side-channel attacks for the crypto-cores is not carefully considered. We are evaluating the side-channel attack scenarios on the proposed crypto-cores and will provide a better design for future work. Secondly, if the protection on I/O paths is not implemented, it could become a weak point in the design against potential attacks. We are fixing it in the next version of our proposed framework. Furthermore, we are considering expanding this work to Network-on-Chip (NoC) systems. Generally, an NoC system boots up a single main CPU first, establishing the secured functions for the network and its other cores. In this model, we must first ensure the security of the main core's boot process. This is precisely what we have achieved through our work. Next, the main core can use the supported crypto-accelerators in this framework to establish Network-on-Chip security.

**Author Contributions:** Supervision, C.-K.P. and T.-T.H.; methodology, T.-T.H., B.K.-D.-N., T.-K.D. and K.-D.N.; investigation, T.-T.H., B.K.-D.-N., T.-K.D., N.T.B. and K.-D.N.; writing—original draft preparation, B.K.-D.-N. and C.P.-Q.; writing—review and editing, N.-T.T., C.-K.P. and T.-T.H. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding

**Data Availability Statement:** Data are contained within the article.

**Acknowledgments:** The VLSI chip in this study has been fabricated through the activities of VLSI Design and Education Center (VDEC), the University of Tokyo, in collaboration with Synopsys, Inc., Cadence Design Systems Inc., Mentor Inc., Rohm Semiconductor (ROHM), and Nippon Systemware Co., Ltd. We also acknowledge the collaboration with Ho Chi Minh City University of Technology (HCMUT), VNU-HCM in facilitating this research.

**Conflicts of Interest:** The authors declare no conflicts of interest.

## References

1. Quarkslab. Introduction to Trusted Execution Environment: ARM's TrustZone. *Retrieved Oct.* **2018**, *8*, 2019.
2. Oracle Corporation. Working with UEFI Secure Boot. Available online: https://docs.oracle.com/en/operating-systems/oracle-linux/secure-boot/sboot-OverviewofSecureBoot.html#sb-overview (accessed on 21 June 2024).
3. Sabt, M.; Achemlal, M.; Bouabdallah, A. Trusted Execution Environment: What It is, and What It is Not. In Proceedings of the IEEE Trustcom/BigDataSE/ISPA (TrustCom), Helsinki, Finland, 20–22 August 2015; Volume 1, pp. 57–64.
4. Intel Corp. *Intel Software Guard Extensions (Intel SGX) Developer Guide*; Intel Corp.: Santa Clara, CA, USA, 2018.
5. Costan, V.; Devadas, S. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086, January 2016. Available online: https://eprint.iacr.org/2016/086 (accessed on 21 June 2024).
6. Costan, V.; Lebedev, I.; Devadas, S. Secure Processors Part I: Background, Taxonomy for Secure Enclaves and Intel SGX Architecture. *Found. Trends® Electron. Des. Autom.* **2017**, *11*, 1–248. [CrossRef]
7. Costan, V.; Lebedev, I.; Devadas, S. Secure Processors Part II: Intel SGX Security Analysis and MIT Sanctum Architecture. *Found. Trends® Electron. Des. Autom.* 2018, *11*, 249–361. [CrossRef]
8. ARM Ltd. *ARM Security Technology: Building a Secure System Using TrustZone Technology*; Technical Report PRD29-GENC-009492C; ARM Ltd.: Cambridge, UK, 2009.
9. Pinto, S.; Santos, N. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Comput. Surv.* **2019**, *51*, 1–36. [CrossRef]
10. Buhren, R.; Werling, C.; Seifert, J.-P. Insecure Until Proven Updated: Analyzing AMD SEV's Remote Attestation. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS), London, UK, 11–15 November 2019; pp. 1087–1099.
11. Baumann, A.; Peinado, M.; Hunt, G. Shielding Applications from an Untrusted Cloud with Haven. In Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), Broomfield, CO, USA, 6–8 October 2014; pp. 267–283.

12. Tsai, C.-C.; Porter, D.E.; Vij, M. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In Proceedings of the USENIX Annual Technical Conference (ATC), Santa Clara, CA, USA, 12–14 July 2017; pp. 645–658.
13. Arnautov, S.; Trach, B.; Gregor, F.; Knauth, T.; Martin, A.; Priebe, C.; Lind, J.; Muthukumaran, D.; O'Keeffe, D.; Stillwell, M.L.; et al. SCONE: Secure Linux Containers with Intel SGX. In Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), Savannah, GA, USA, 2–4 November 2016; pp. 689–703.
14. Ferraiuolo, A.; Baumann, A.; Hawblitzel, C.; Parno, B. Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. In Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), Shanghai, China, 28–31 October 2017; pp. 287–305.
15. Linaro Ltd. *Open Portable Trusted Execution Environment*; Linaro Ltd.: Cambridgeshire, UK, 2021.
16. Brasser, F.; Gens, D.; Jauernig, P.; Sadeghi, A.-R.; Stapf, E. SANCTUARY: ARMing TrustZone with User-space Enclaves. In Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, USA, 24–27 February 2019; pp. 1–15.
17. Kaplan, D. Protecting VM Register State with SEV-ES. *White Paper* 17 February 2017. Available online: https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/Protecting-VM-Register-State-with-SEV-ES.pdf (accessed on 21 June 2024).
18. Sev-Snp, A.M.D. Strengthening VM Isolation with Integrity Protection and More. *White Paper* January 2020. Available online: https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf (accessed on 21 June 2024).
19. Hex Five Security, Inc. *MultiZone Hex-Five Security*; Hex Five Security, Inc.: Redwood Shores, CA, USA, 2024.
20. Costan, V.; Lebedev, I.; Devadas, S. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In Proceedings of the 25th USENIX Security Symposium (USENIX Security 16), Austin, TX, USA, 10–12 August 2016; pp. 857–874.
21. Weiser, S.; Werner, M.; Brasser, F.; Malenko, M.; Mangard, S.; Sadeghi, A.-R. TIMBER-V: Tag-Isolated Memory Bringing Fine-Grained Enclaves to RISC-V. In Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 24–27 February 2019; pp. 1–15.
22. Bahmani, R.; Brasser, F.; Dessouky, G.; Jauernig, P.; Klimmek, M.; Sadeghi, A.-R.; Stapf, E. CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In Proceedings of the USENIX Security Symposium (USENIX Security), Virtual Event, 11–13 August 2021; pp. 1073–1090.
23. Lee, D.; Kohlbrenner, D.; Shinde, S.; Asanovic, K.; Song, D. Keystone: An Open Framework for Architecting Trusted Execution Environments. In Proceedings of the European Conference on Computer Systems (EUROSYS), Heraklion, Greece, 27–30 April 2020; pp. 1–16.
24. He, F.; Zhang, H.; Wang, H.; Xu, M.; Yan, F. Chain of Trust Testing Based on Model Checking. In Proceedings of the International Conference on Networks Security, Wireless Communications and Trusted Computing (NSWCTC), Wuhan, China, 24–25 April 2010; Volume 1, pp. 273–276.
25. AMD Inc. Inside a Deeply Embedded Security Processor. In Proceedings of the Black Hat USA, Virtual Event, 1–6 August 2020; AMD Inc.: Santa Clara, CA, USA, 2020.
26. ARM Ltd. *ARM Security IP: CryptoCell-700 Family*; ARM Ltd.: Cambridge, UK, 2017.
27. Intel Corp. *Intel Active Management Technology (AMT) Developers Guide*; Intel Corp.: Santa Clara, CA, USA, 2024.
28. Rambus, Inc. *Security CryptoManager Provisioning*; Rambus, Inc.: Sunnyvale, CA, USA, 2022.
29. lowRISC CIC. OpenTitan. Available online: https://github.com/lowRISC/opentitan (accessed on 21 June 2024).
30. *ISO/IEC 11889-1:2015*; Information Technology—Trusted Platform Module Library—Part 1: Architecture. ISO/IEC: Geneva, Switzerland, 2015.
31. Furtak, A.; Bulygin, Y.; Bazhaniuk, O.; Loucaides, J.; Matrosov, A.; Gorobets, M. BIOS and Secure Boot Attacks Uncovered. In Proceedings of the Ekoparty Security Conference, Buenos Aires, Argentina, 29–31 October 2014; pp. 1–79.
32. Cui, E.; Li, T.; Wei, Q. RISC-V Instruction Set Architecture Extensions: A Survey. *IEEE Access* **2023**, *11*, 24696–24711. [CrossRef]
33. Hoang, T.T.; Duran, C.; Serrano, R.; Sarmiento, M.; Nguyen, K.D.; Tsukamoto, A.; Suzaki, K.; Pham, C.K. Trusted Execution Environment Hardware by Isolated Heterogeneous Architecture for Key Scheduling. *IEEE Access* **2022**, *10*, 46014–46027. [CrossRef]
34. PHAM Laboratory. TEE Hardware Platform. Available online: https://github.com/uec-hanken/tee-hardware (accessed on 21 June 2024).
35. Serrano, R.; Duran, C.; Hoang, T.-T.; Sarmiento, M.; Nguyen, K.-D.; Tsukamoto, A.; Suzaki, K.; Pham, C.-K. A Fully Digital True Random Number Generator with Entropy Source Based in Frequency Collapse. *IEEE Access* **2021**, *9*, 105748–105755. [CrossRef]
36. Dworkin, M.J. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. August 2015. Available online: https://www.nist.gov/publications/sha-3-standard-permutation-based-hash-and-extendable-output-functions?pub_id=919061 (accessed on 21 June 2024).
37. *FIPS-197*; Advanced Encryption Standard (AES). NIST Standard: Gaithersburg, MD, USA, November 2001.
38. Krawczyk, H.; Bellare, M.; Canetti, R. RFC2104: HMAC: Keyed-Hashing for Message Authentication. February 1997. Available online: https://dl.acm.org/doi/abs/10.17487/RFC2104 (accessed on 21 June 2024).
39. Nir, Y.; Langley, A. RFC8439: ChaCha20 and Poly1305 for IETF Protocols. June 2018. Available online: https://datatracker.ietf.org/doc/rfc8439/ (accessed on 21 June 2024).
40. lowRISC CIC. IBex RISC-V Core. Available online: https://github.com/lowRISC/ibex (accessed on 21 June 2024).

41. Barker, E.; Roginsky, A.; Davis, R. *Recommendation for Cryptographic Key Generation*; Technical Report; National Institute of Standards and Technology (NIST): Gaithersburg, MD, USA, 2020.
42. SiFive, Inc. *SiFive TileLink Specication*; SiFive, Inc.: Santa Clara, CA, USA, 2019.
43. ARM. *AMBA AXI and ACE Protocol Specification*; Technical Report ARM IHI 0022H.c; ARM: Cambridge, UK, 2021.
44. Hofmann, O.S.; Kim, S.; Dunn, A.M.; Lee, M.Z.; Witchel, E. InkTag: Secure Applications on an Untrusted Operating System. *ACM SIGPLAN Not.* **2013**, *48*, 265–278. [CrossRef]
45. Stratify Labs. *Dhrystone Benchmarking on MCUs*; Stratify Labs: Highland, UT, USA, 2019.
46. Kumar, V.B.Y.; Chattopadhyay, A.; Yahya, J.H.; Mendelson, A. ITUS: A Secure RISC-V System-on-Chip. In Proceedings of the IEEE International System-on-Chip Conference (SOCC), Singapore, 3–6 September 2019; pp. 418–423.
47. Yahya, J.H.; Wong, M.M.; Pudi, V.; Bhasin, S.; Chattopadhyay, A. Lightweight Secure-Boot Architecture for RISC-V System-on-Chip. In Proceedings of the International Symposium on Quality Electronic Design (ISQED), Santa Clara, CA, USA, 11–13 March 2019; pp. 216–223.
48. Nasahl, P.; Schilling, R.; Werner, M.; Mangard, S. HECTOR-V: A Heterogeneous CPU Architecture for a Secure RISC-V Execution Environment. In Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIA CCS), Hong Kong, China, 7–11 June 2021; pp. 187–199.
49. SiFive, Inc. *Securing the RISC-V Revolution*; SiFive, Inc.: Santa Clara, CA, USA, 2019.