Secure Code Execution: A Generic PUF-Driven System Architecture

Stephan Kleber[1(✉)], Florian Unterstein[2], Matthias Hiller[2], Frank Slomka[3], Matthias Matousek[1], Frank Kargl[1], and Christoph Bösch[1]

[1] Institute of Distributed Systems, Ulm University, Ulm, Germany
{stephan.kleber,matthias.matousek,frank.kargl,christoph.bosch}@uni-ulm.de
[2] Fraunhofer Institute for Applied and Integrated Security (AISEC), Munich, Germany
{florian.unterstein,matthias.hiller}@aisec.fraunhofer.de
[3] Institute of Embedded Systems/Real-Time Systems, Ulm University, Ulm, Germany
frank.slomka@uni-ulm.de

Abstract. In his invited talk, joint between CHES 2016 and CRYPTO 2016 on the Future of Embedded Security, Paul Kocher suggested to move the security into chips because hardware is the lowest level and thus security can not be compromized by a lower layer. In this paper, we propose a generic PUF-driven secure code execution architecture that employs instruction-level code encryption. Our design foresees a tight integration of a Physically Unclonable Function (PUF) and the decryption of encrypted program code directly inside the processor's instruction pipeline to avert revealing keys or decrypted code in externally accessible registers or memory. The architecture prevents code-injection by executing only code encrypted for individual target CPUs, has an adaptable impact on performance, and requires only minor changes to the software development process. Our PUF-based code encryption defends also from reverse engineering attempts and enforces IP protection. A proof-of-concept implementation demonstrates the feasibility of our proposed architecture.

Keywords: Secure Execution Environment · PUF · FPGA

1 Introduction

Secure and reliable execution of code is of high importance for classical networked systems in the Internet but even more for networked control systems for critical processes. Such Cyber-Physical Systems can be found in industrial plants, in home-automation and in connected cars. In order to avoid negative consequences like misguided process control, one needs to ensure that only valid and unaltered code is executed on such embedded devices. At the same time, such code often contains intellectual property (IP) which makes it necessary to protect the code from extraction and reverse engineering to avoid leaking of IP.

L. Chen et al. (Eds.): ISC 2018, LNCS 11060, pp. 25–46, 2018.
https://doi.org/10.1007/978-3-319-99136-8_2

Code injection, especially when performed remotely, is one of the most effective strategies for malicious attackers. Stuxnet, for example, exploited such a remote code execution vulnerability (CVE-2008-4250) to infect remote machines [18]. Since the popular phrack article "Smashing the Stack for Fun and Profit" [24] by Aleph One in 1996, which described simple stack buffer overflows, new detection and prevention techniques like stack canaries or non-executable stacks have been proposed just to soon thereafter be circumvented by more sophisticated attack techniques like Return-Oriented Programming (ROP). Now, more than two decades later, it is still an open security challenge to effectively prevent injection of unauthorized code into an execution environment. A *Secure Execution Environment* (SEE) protects against such *code injection* to prevent malicious actions to be executed inside the environment (ensuring *code integrity*) and additionally, often prevents genuine code from getting extracted out of its execution environment to prevent reverse engineering (ensuring *code confidentiality*). Thereby, an SEE not only contributes to enhanced resilience of an otherwise vulnerable platform against malicious manipulation, but also helps to protect the IP of software and hardware manufacturers. Based on our threat model (Sect. 2), we present our generic PUF-driven secure code execution architecture which allows secure execution of encrypted programs where programs are encrypted per processor instance (Sect. 3). Code that is not properly encrypted for a specific device will not execute correctly, triggering faults. Thus, an attacker will not be able to produce and inject valid code that will exhibit a desired malicious behavior. As an additional effect, relying on a PUF for re-generating decryption keys implements a strict hardware-binding, where code cannot be transferred between devices [11,26]. At the same time, the architecture will effectively prevent reverse engineering of programs stored in such an embedded system as all such code is encrypted and will only be decrypted directly inside the execution pipeline on a per-instruction basis. Thereby, our design minimizes exposure of decryption keys or decrypted instructions to isolated dedicated registers. The proposed design is comparatively lightweight with a well-defined minimum trusted computing base (TCB) that does not include external or internal memory or caches. The design can be implemented by a standard production process. While implementing the architecture, use-case-specific parameters of freely exchangeable building blocks allow for customization which we discuss in Sect. 4. Based on a proof-of-concept implementation (Sect. 5) we investigate the feasibility of our architecture and its overhead in Sect. 6. The analysis of the state-of-the-art in Sect. 7 shows where our architecture outperforms comparable earlier approaches. In Sect. 8, we conclude that our design constitutes a major step towards code integrity and code confidentiality in embedded devices. Based on the lessons learned, we present a list of extensions and enhancements that might be of interest in future work.

2 Threat Model

Our main goal is to protect from code injection attacks, starting from simple execution of untrusted code but also extending to injection attacks like stack

buffer overflows and even advanced attacks like ROP. As additional goal, we aim for code confidentiality, and thus, reliably prevent reverse engineering of code and IP infringements.

For protection from malicious *code injection*, we prevent any non-authorized code to be executed – or rather lead to any behavior that the attacker can control or predict. Beyond, manipulations of the control flow of a process needs to be prevented, as some advanced attacks like ROP can achieve their malicious goals without actually injecting additional instructions.

We consider three types of attacks under this term:

1. Attacks where unauthorized code should be executed in a new process or thread of its own using whatever execution procedure is foreseen on the target platform.
2. We consider attacks where an attacker tries to exploit software implementation flaws like susceptibility to buffer overflows in order to inject self-crafted code into an existing process or thread and manipulate the program flow to execute this code.
3. There are attacks that do not involve any additional instructions but only change the program flow in a way to exhibit new and unintended behavior. ROP is a prime example of this.

Attacks two and three may be performed remotely or locally via the intended input methods of legitimate software running on the target processor. All these attacks are software-centric, however, we do not exclude hardware attackers in our discussion. Hardware attackers have physical access to the processor, its memory and peripherals, and might try to inject code by direct manipulation of memory and any data and instruction cache off- and on-chip. Fault-injection attacks also fall in this category.

In the case of *code confidentiality*, an attacker wants to gain additional knowledge on the application code deployed on a system, e.g., as part of a reverse engineering effort. The system may be under physical control of the attacker, like often the case in an embedded or mobile system running a supplier's confidential firmware. We assume that all off-chip bus lines can be probed via openly accessible chip pins and that all external memory can be read out.

A more sophisticated attacker may even be able to use hardware probes to read-out from or inject bits into on-chip buses [17]. Beyond, the chip design and layout may be known to the attacker. This results in an attacker capable to obtain memory contents via methods like linear code extraction and direct memory probing. In addition, invasive tampering like glitching or fault injection on bus lines may also be possible. While an attacker with such advanced capabilities may be extremely hard to defend from, the goal of our architecture is to minimize the trusted computing base, i.e., the area of the chip and the duration where such confidential information is exposed.

We note that further attacks like denial-of-service (DoS) and hardware side-channels are not specifically considered within our approach. Complementary research exists, proposing countermeasures to side-channel attacks.

Ascend [10], for example, is specifically designed to hamper side-channel attackers by obfuscating memory access patterns, power consumption and temperature analyses.

3 Our Architecture for Secure Code Execution

Execution of program code typically happens detached from its development. Therefore, our architecture distinguishes between a development machine (*user system*) and the execution environment (*target system*). A central advantage of our architecture is that only encrypted program binaries can be executed which are bound to a specific processor and which can be decrypted and run only by that specific processor. To be able to use the benefits of the PUF at execution time without the need to have it available at development time, we separated the encryption of the program binary from binding this binary to the PUF. Therefore, two cryptographic processes have to be discerned: The encryption and decryption of the binary itself is performed by a random access cipher (e.g., XTS [1] or CTR mode), without the use of the PUF (Fig. 1(a), Step 2 and (b), Step 3). In a separate process, the key used for this operation is encrypted and decrypted involving the PUF (Fig. 1(a), Step 5 and (b), Step 2).

The architecture of the target system consists of a generic RISC CPU, a instruction decryption module, and a PUF module. Our CPU uses fixed-length instructions of 32 bit each. The instruction decryption module and the PUF module are closely coupled with the CPU's execution pipeline.

During program execution, the cryptographic **instruction decryption module** reads the encrypted binary and decrypts it, (Fig. 1(b), Step 3) instruction by instruction, during the instruction fetch (IF) stage. This is performed by

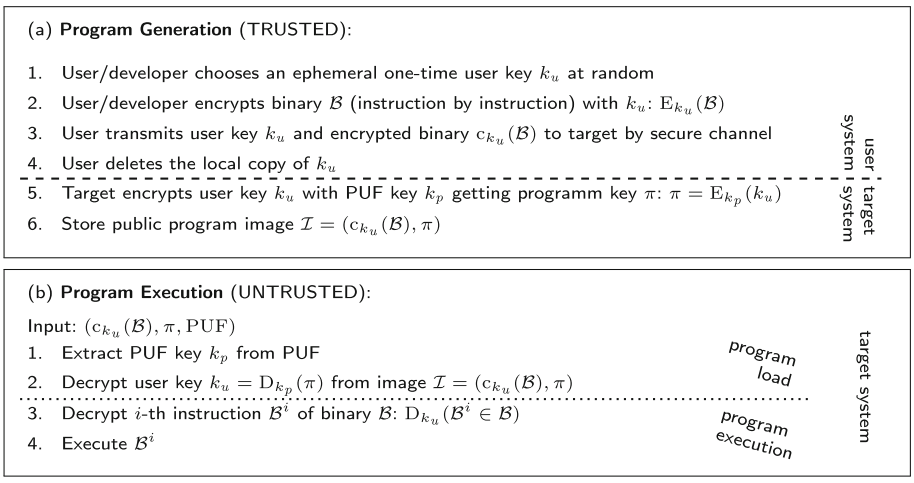


Fig. 1. Program encryption and execution process.

a random access block cipher. Decrypted instructions are directly forwarded to the instruction decode stage (Fig. 1(b), Step 4) and continue their normal walk through the pipeline. The binary-specific key k_u for the block cipher is secured through the use of the **PUF module**, ensuring that any encrypted program can be decrypted only on the device instance it was created for.

Notation: Encryption and decryption operations of plaintext p and ciphertext c with key k: $\mathrm{E}_k(p)$ and $\mathrm{D}_k(c)$; Encrypted entity or ciphertext c of plaintext p with key k: $\mathrm{c}_k(p)$ is output of $\mathrm{E}_k(p)$; Cryptographic hash function: $\mathrm{H}(\cdot)$.

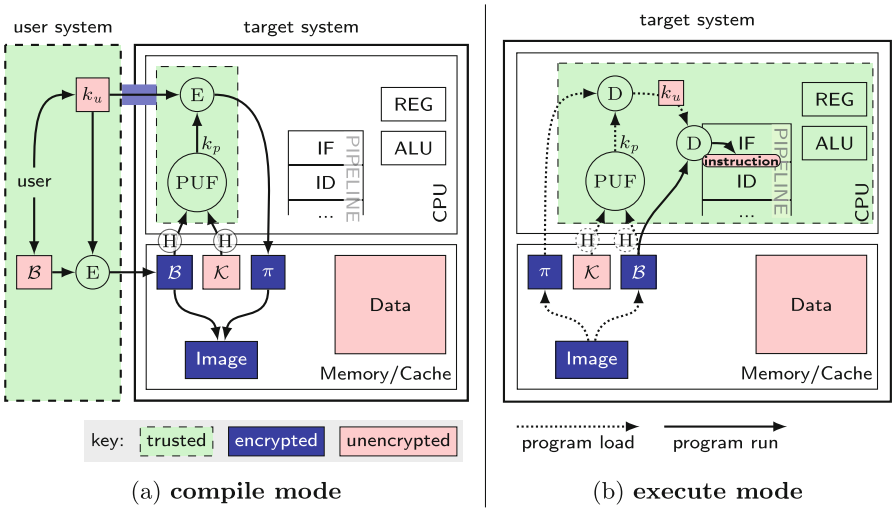
We call the development machine *user system* and consider it trusted. To generate a program to be executable on the target system, the user compiles confidential source code into a binary using a default compiler, e.g., gcc. The user arbitrarily chooses an ephemeral one-time key k_u (Fig. 1(a), Step 1) and encrypts this binary, instruction by instruction (Fig. 1(a), Step 2). The encrypted binary is then transferred alongside with k_u to the target system through a secure channel (Fig. 1(a), Step 3). Afterwards, the users local copy of k_u can be deleted (Fig. 1(a), Step 4). On the target system, the user key k_u to encrypt the binary gets itself encrypted by the PUF-generated key k_p (Fig. 1(a), Step 5) yielding the $\pi = \mathrm{c}_{k_p}(k_u)$. The PUF is available on the target system and the PUF's output can therefore be used as key for this second cipher step.

Each time the program is loaded, this second encryption process is reversed by decrypting the program-specific key into k_u (Fig. 1(b), Step 1 and 2). This k_u is used during execution of the according binary to decrypt the individual instructions by the instruction decryption module and is kept directly inside the decryption module. The choice of a suitable cipher mode has to take into account the requirements induced by the program flow. Program flow must generally allow for jumps to enable branching and loops, so random access within the instruction stream is mandatory. Only specific random access modes like CTR mode of a symmetric cipher fulfill the requirement of random access. We discuss security implications, in Sect. 4, and Sect. 6.1.

The roles of the components of the architecture during program generation and program execution are explained in the following sections. Generation and execution processes are dependent on several entities, besides k_u. Foremost, this is the binary $\mathcal{B}$, resulting from the user's compilation of source code. $\mathcal{B}$ is generic and needs not to be recompiled for use with another processor instance. The binary, encrypted with the key k_u, is denoted $\mathrm{c}_{k_u}(\mathcal{B})$. k_u and $\mathcal{B}$ have to remain confidential, whereas $\mathrm{c}_{k_u}(\mathcal{B})$ may be public.

3.1 Program Generation

Program generation takes place in a trusted environment (Fig. 2a). The user chooses an ephemeral one-time key k_u for the block cipher and encrypts the compiled $\mathcal{B}$ with a random access cipher. A program flow is structured in basic blocks. A basic block is a sequence of instructions with exactly one entry point and one exit point, and no branches in between. Each basic block is encrypted to later be decrypted by the hardware instruction decryption module.



(a) **compile mode** (b) **execute mode**

Fig. 2. Our proposed architecture

The program encryption itself is not critical for runtime performance and requires no hardware support. Particularly, it does not need the PUF or any key material from it. We currently implement it as standalone tool, processing the compiled binary in software, but it may be included directly into the compiler.

To finally bind $\mathcal{B}$ to a hardware instance, inherent properties of the PUF are utilized. This part of the process of program generation is shown in Fig. 2a. The encrypted binary $c_{k_u}(\mathcal{B})$ (depicted in blue) and k_u are transmitted to the target system. Since encrypted, $c_{k_u}(\mathcal{B})$ may be public, but k_u must remain confidential. To transmit k_u securely to the PUF module (depicted as blue "tunnel"), a dedicated cable connection can be used during deployment in a secure production environment. For over-the-air (OTA) deployment, a secure channel (e.g., TLS, SSH) is required. The PUF module then generates a cryptographic key k_p taking the encrypted binary $c_{k_u}(\mathcal{B})$ and the security kernel $\mathcal{K}$ as additional input to prevent their modification. A security kernel $\mathcal{K}$ comprises all basic software components required for secure boot of the system (see also Sect. 3.3).

To protect the user key k_u—required for program execution—it is encrypted on the target system using k_p. The encrypted representation $\pi = c_{k_p}(k_u)$ can be disclosed publicly. It is then stored together with the encrypted binary to form the *program image* $(c_{k_u}(\mathcal{B}), \pi)$ in publicly shared memory. This scheme has the advantage that the user does not require access to the target hardware for preparation of a binary to be executed on it while retaining the security properties of the PUF.

3.2 Program Decryption and Execution

To minimize the TCB, i.e., the parts of the processor required to be trusted, the decryption module is entangled with the instruction fetch (IF) stage of the processor's pipeline as shown in Fig. 2b. Encrypted instructions enter the stage and are decrypted immediately before proceeding to the instruction decode (ID) stage. Thus, it is possible to encrypt $\mathcal{B}$ on a per instruction level.

The *nonce and counter generator* prepares the counter as input to the block cipher. It concatenates the base address of each basic block concatenated with an instruction counter inside the block. Whenever there is a jump, a new base address is set and the counter is reset to 0. That way, jumps can only target a start of a basic block, otherwise, the decryption result will be invalid. The module pre-computes keystream blocks by incrementing the counter value and allows modularity in the control flow of the executed code. The end of a basic block is reached when either a jump occurs or the execution linearly continues into the next basic block. In either case, the processor detects the new block and reinitializes the decryption accordingly. The implementation has to take care of resetting the encryption and stalling the processor whenever necessary.

3.3 The Role of the PUF

Since encrypted under k_u, any program image $\mathcal{I} = (\mathrm{c}_{k_u}(\mathcal{B}), \pi = \mathrm{c}_{k_p}(k_u))$ for a processor instance may be public. However, the secret key k_u is required in the target system's instruction-fetch (IF) logic during execution to generate the decrypted keystream. This is depicted in Fig. 2b, showing how the PUF module decrypts k_u from the image $\mathcal{I}$ with help of the PUF key k_p and then forwards k_u directly to the decryption. However, a processor instance should not need to store any information about an image; thus, $\mathcal{I}$ has to be self-contained. This is accomplished by recovering k_u from the encrypted π inside the program image and k_p each time a binary is loaded for execution. Only the very same processor instance can generate the correct k_p to restore $k_u = \mathrm{c}_{k_p}(\pi)$. k_p has to be derived from the target system's PUF, using hashes of the encrypted binary and the security kernel $(\mathrm{H}(\mathrm{c}_{k_u}(\mathcal{B})), \mathrm{H}(\mathcal{K}))$ as parameters. This way k_p is not only bound to the device but also to the integrity of the program binary $\mathcal{B}$ and the security kernel $\mathcal{K}$. If $\mathcal{B}$ or $\mathcal{K}$ is modified, the derived key k_p changes and cannot be used to decrypt the user key k_u anymore. A security kernel $\mathcal{K}$, like the boot loader or the operating-system core, guarantees secure access to system resources.

To securely combine all parameters of the PUF module into one key, we propose a key derivation function (KDF) like presented by Krawczyk [16]. From the PUF output s_p and the key-derivation parameters, the PUF module calculates:

$$k_p = \mathrm{KDF}(s_p, \mathrm{H}(\mathrm{c}_{k_u}(\mathcal{B})), \mathrm{H}(\mathcal{K}))$$

During **loading** of $\mathcal{B}$, k_p is derived once and used to recover $k_u = \mathrm{c}_{k_p}(\pi)$. In the IF stage while **executing**, the ith instruction $\mathcal{B}^i$ of $\mathcal{B}$ gets decrypted by:

$$\mathrm{Cipher}_{k_u}(nonce||i) \oplus \mathrm{c}_{k_u}(\mathcal{B}^i) = \mathcal{B}^i$$

Using a PUF, we do not need any secure storage across all memory hierarchies in our system and thereby minimize the secure and trusted computing base.

4 Customization for an Implementation

A number of security-relevant building blocks exist in our architecture, for which concrete crypto-primitives have to be chosen during an implementation. The building blocks that an implementation has to define are: (1) the PUF-type, (2) the KDF in the PUF-module, (3) the block-cipher for the user-key en- and decryption, (4) the cipher-mode for the instruction decryption, and (5) the block-cipher for the instruction decryption.

The selection of crypto primitives is depending on the desired level of security and performance for the use case of an implementation. A large number of options exist for this customization since almost any combination of block-ciphers, random-access cipher modes, KDFs, and PUFs is possible. Without loss of generality, we provide examples for a suitable selection of crypto-primitive sets for each building block in Table 1. All alternatives are a trade-off between the level of security and performance or chip-area. Hardware implementations of ciphers with high throughput, typically, need more area. Low latency ciphers reduce the performance impact, while they, typically, are more susceptible to attacks [21].

Table 1. Examples of crypto-primitives for an implementation.

Building block	Example 1	Example 2	Example 3
PUF-type	Ring-Oscillator	SRAM	Bistable-Ring
Key derivation (KDF)	SHA-256 of concatenated keys	HKDF	HMAC-SHA-256
Block-cipher (user-key)	AES-128	PRINCE	AES-256
Cipher-mode (instr. decr.)	CTR	XTS	LRW
Block-cipher (instr. decr.)	AES-128	PRINCE	SIMON

The throughput of block-ciphers in hardware can be increased by paralleliz-ing cores, but occupying more area. Alternative low-latency block ciphers, like PRINCE [6] or SIMON [4], may improve performance and reduce area overhead, at the cost of reduced security as evaluated by Maene et al. [21]. Alternative random access cipher modes are understood from disc encryption, where an encryption block is not chained with the previous plain-text block. Candidates for the usage in our architecture are the block-cipher modes devised by *Liskov, Rivest, and Wagner* (LRW) [20], being an improved security-complexity trade-off compared to CTR, or the *XEX-based tweaked-codebook mode with ciphertext stealing* (XTS) [1].

5 Proof-of-Concept Implementation

To demonstrate the feasibility of our architecture, we implemented a proof-of-concept (PoC) of our design capable of creating and executing encrypted standalone program images. We call our PoC implementation the *Secure Execution PUF-based Processor* (SEPP). For the cryptographic primitives for the customizable parts, we decided for a conservative selection of thoroughly tested implementations of well-known algorithms (Table 1, Example 1).

SEPP is based on an OR1200[1] processor, an implementation of the OpenRISC OR1000 architecture. It is a popular open-source RISC architecture with a 32 bit wide fixed-length instruction set and a five stage, single issue pipeline. We chose an architecture with fixed-length instructions in order to be able to always map whole instructions to cipher blocks without overlap and alignment issues. The key parameters of our system are: Architecture: 32 bit RISC; clock frequency: 50 MHz; instruction cache: 8 Kbyte, 1-way direct-mapped; data cache: 8 Kbyte, 1-way direct-mapped; Memory: 128 MB DDR2 SDRAM.

We extended the OpenRISC Reference Platform (ORPSoC) by two modules according to our architecture: the PUF module and the instruction decryption module. We deployed our design on a Digilent Atlys Board powered by a Xilinx Spartan-6 LX45 FPGA. As development environment we used Xilinx ISE 14.7. For synthesis, placement, and routing parameters, as well as FPGA-dependent settings, like operating frequency, we used the defaults distributed with the ORPSoC make files. This enables us to directly compare SEPP to ORPSoC.

The RO PUF [23] implementation we utilized generates a single fixed response by comparing the oscillating frequencies of ring oscillator pairs. To counter the noise of the PUF output, the C-IBS fuzzy extractor [12] (see Sect. 7) takes the PUF output and creates helper data to reliably recreate a response. The corresponding helper data can be stored in memory since an attacker gains no advantage from it. The output of the C-IBS fuzzy extractor we call s_p, the reliable, embedded *PUF secret*. The PUF module encapsulates the RO PUF with the C-IBS fuzzy extractor and a low area AES core in ECB mode for encryption and decryption of k_u on load of the program image. Our PoC implements hashing and combining of the encrypted binary $\mathrm{H}(\mathrm{c}_{k_u}(\mathcal{B}))$ and the security kernel $\mathrm{H}(\mathcal{K})$ in software. The hardware PUF module then combines this result with the PUF secret s_p to create the PUF key k_p. For s_p to remain confidential, there is no external interface to access the outputs of the PUF or the fuzzy extractor.

The CPU is interfaced with the PUF module over a dedicated class of special purpose registers. Those include configuration and status registers as well as the input for the combined hashes $(\mathrm{H}(\mathrm{c}_{k_u}(\mathcal{B})), \mathrm{H}(\mathcal{K}))$. It is important to note that in program execution mode, k_u is directly forwarded to the decryption module over a separate connection and is not visible on any system bus.

The instruction decryption module is integrated in the processor's instruction fetch stage and has no dedicated interface other than the input of k_u from the PUF module. As shown in Fig. 3, the decryption module evaluates the Program

[1] http://opencores.org/or1k/OR1200_OpenRISC_Processor, accessed 03/13/2017.

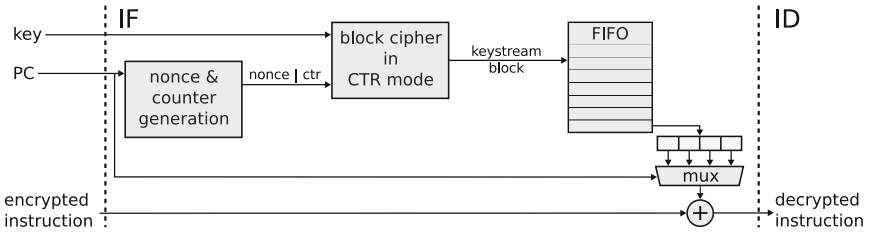


Fig. 3. Simplified structure of the decryption module

Counter (PC) to detect branches and sets the nonce and counter accordingly. The program flow from one basic block to the other is marked by a custom instruction, if the block has no preceding branch instruction. These marker instructions are added prior to the encryption on the user system by a small compiler add-on. Some instructions take more than one processor cycle to execute. Thus, a first-in, first-out (FIFO) buffer is added to store the produced blocks of the keystream until they are needed by the processor. Since a keystream block produced by the AES-128 cipher can cover four 32 bit instructions of the CPU, an additional multiplexer following the FIFO is used. It selects the correct part of the keystream block depending on the current PC. This portion of the keystream block is XORed with the incoming encrypted instruction and the decrypted instruction is forwarded to the instruction decode stage.

The AES decryption core, for instruction decryption in CTR, should provide a throughput sufficiently high to allow sequential code to be decrypted and executed without stalling. We evaluated several freely available 128 bit-key AES cores and selected one[2] with enough throughput (in *keystream blocks per cycle*) for uninterrupted execution.

For an optimal trade-off between FPGA resources, lowest latency, and throughput, we designed a parallel combination of four 13-cycle encrypt-only AES cores. In CTR mode, the AES core has to support only encryption. Using this design, the processor needs to be stalled only once at the beginning of each basic block for 13 cycles until the first portion of keystream for a basic block is computed.

The whole SEPP implementation on the Spartan-6 FPGA requires 20,643 Slice LUTs overall and 11,321 Registers, compared to 12,542 LTUs and 6,752 Registers of the baseline ORPSoC. Most of the additional Slices are required for the PUF and a significant amount for the AES cores. The control logic and interfaces require only few additional resources. The required resources seem to be a significant overhead, but recall that all building blocks are exchangeable and nothing except for SEPP has to run on the device.

The Universal Bootloader *Das U-Boot* (or U-Boot)[3] is used as our software platform. It was modified to implement the functionality of the target system

[2] http://opencores.org/project,aes_core, accessed on 03/13/2017.
[3] http://www.denx.de/wiki/U-Boot, accessed on 03/13/2017.

security kernel, i.e., the generation and execution of encrypted program images in interaction with the PUF module. It provides a simple command line interface to the user, for which we added custom commands. RSA public key cryptography is used by U-Boot to establish a secure communication channel between the user system and the target system. The KDF calculation for the PUF is implemented as part of our security kernel.

Program encryption is independent from the PoC hardware and is handled at the user system. In our case, the user system is a common Linux system for which we developed helper tools that analyze the compiled binary, encrypt the code, and package it together with the RSA-encrypted k_u in an U-Boot image. The user transfers such an encrypted program image to the PoC system over an Ethernet connection. There π is generated and replaces the k_u embedded in the image. This image can now be securely made public as it only contains encrypted code and the encrypted key k_u in form of the public π. The encrypted image is tied to this exact hardware instance and only this processor with its unique PUF is able to decrypt and execute it.

6 Discussion

6.1 Security

Through the use of a PUF, there is no unsecured key material that needs to be stored persistently. Moreover, our architecture is designed to minimize the TCB, so only the PUF, registers, pipeline, and the ALU need to be trusted. In contrast, previous approaches needed to trust all on- and off-chip bus lines and all memory and caches, as presented in Sect. 7. However, invasive hardware attacks against our architecture, during program execution, could allow for access to bus lines between PUF, registers, pipeline and ALU or those components themselves. The most prevalent hardware attacks on processors currently target off- and on-chip memory bus lines [17]. Although sophisticated and very costly, these attacks allow for read-out and injection—mostly fault injection—of data bit by bit. This class of attacks is countered with meshes and other sophisticated hardware countermeasures, e.g., in smartcard processors. The authors of XOM, AEGIS, OASIS, and Ascend note that a variety of methods exist to prevent or impede hardware tampering, e.g., probing or fault-injection. We agree that such means will become necessary as soon as this class of attacks should become feasible for processors in CMOS technique. In such a case, costly hardware attack countermeasures can be applied to our architecture much easier, compared to securing the whole SoC, including caches and any kind of memory. This follows from the small chip area to be trusted, compared with previous designs. Figure 2a and 2b show the trusted area in the design in green color. This provides an unprecedented option to reduce production cost for this kind of hardware security measure. To confirm the security of our design, we consider both parts of the threat model (Sect. 2) *code confidentiality* and *injection prevention* separately.

Code Confidentiality. To achieve code confidentiality, the binary image is encrypted. The immediate key necessary to decrypt the binary is k_u. Knowing a specific $\pi = \mathrm{E}_{k_p}(k_u)$ and having available the encrypted binary $\mathrm{c}_{k_u}(\mathcal{B})$ and the security kernel $\mathcal{K}$ as context input for key derivation [16], k_u can be recovered using the correct PUF instance to derive k_p. For key derivation, we require the KDF to be constructed using keyed cryptographic hash functions $\mathrm{H}(\cdot)$. Their second pre-image resistance prevents finding another binary that produces the same output of the KDF. The KDF also breaks the direct link between user inputs and k_p which could otherwise be exploited. Thereby, it prevents known-plaintext-attacks on any of the inputs and possibly the output of the en- or decryption. In addition, a KDF is resilient against length extension attacks. Only k_u and k_p need to remain confidential. k_p never leaves the PUF module, whereas k_u originates externally. Therefore, the confidentiality of a program is determined by the confidentiality of k_u outside of the processor's instruction decode module.

The only time where k_u is required outside of the processor is for the user to encrypt the binary after compilation. Provided the adversary model and scope of our approach, the only attack vector arises during the necessary transfer of k_u from the user machine to the target machine. Therefore, we require a secure channel between those two endpoints. This can be ensured during deployment. Only a low bandwidth is necessary for this channel, since the key is small compared to the binary. We assume that an update mechanism implemented in trusted software is possible and propose an approach to be explored in follow-up work in Appendix A. The user machine itself must be secured and trusted. This can be addressed by known means for host security and is out of the scope of this work.

Injection Prevention. The prevention of code injection is accomplished by executing only correctly encrypted code. k_p is required to recover k_u necessary to en- and decrypt code of a specific program. Thus, security relies only on the confidentiality of k_p. Provided the threat model, no other kind of trust needs to be assumed. k_p can only be generated by the correct PUF instance with an unmodified binary $\mathcal{B}$ and an unmodified security kernel $\mathcal{K}$ as input. Blocking the key generation data path in the processor—for ultimate security demands by a hard-wired switch, otherwise by a privileged instruction—the generation of a new valid $\pi = \mathrm{E}_{k_p}(k_u)$ for any program can be prevented. Since k_p is dependent on $\mathrm{c}_{k_u}(\mathcal{B})$, no new code using a valid π can be generated. This way, arbitrary code injections, like buffer overflows, are effectively prevented. Because of the per-basic-block encryption, the architecture severely limits an attacker's ability to construct useful return oriented programming (ROP) gadgets. Therefore, it effectively prevents malicious control flow manipulation, even for code that is vulnerable on a conventional architecture.

Our SEPP implementation does not use any signatures of individual instructions in $\mathcal{B}$ to check code integrity. Instead it relies on the failure to decrypt a *valid* CPU-instruction at runtime, when an attacker injects a guessed encryption of an

instruction. In the event of an attack, this exception can gracefully be dealt with by defining an exception handler in the processor that returns the control flow back to the trusted $\mathcal{B}$ or even $\mathcal{K}$. In the context of instruction set randomization, Barrantes et al. [3] established that the guessing of a valid instruction is a sufficiently large obstacle for an attacker: For the PowerPC architecture, Barrantes et al. conclude that escapes are successful in less than 5% of all cases. Since OpenRISC has a smaller instruction set than the PowerPC, this makes it even less likely for random bytes to be interpreted as a valid instruction. We estimate that a typical binary contains 10% branching targets to which the execution can jump. Combined with the findings of Barrantes et al., the probability for a successful instruction escape drops to below $10\% \cdot 5\% = 0.5\%$. Thus, to targetedly inject 5 instructions in a row would succeed in only $3 \cdot 10^{-12}$ of all cases. To run a meaningful attack, the attacker needs to control more than one instruction in sequence. Therefore, only relying on the decryption of invalid instructions is superior to validated encryption, under most *practical* circumstances, when taking runtime performance into account, due to the high probability that executing random bytes results in an exception [3]. For high security applications, the used cipher can be replaced by a lightweight authenticated and verifiable encryption scheme, e.g., ALE [5].

To prevent the recovery of the true k_u from π for a tampered-with binary, the PUF-based key derivation requires $\mathrm{c}_{k_u}(\mathcal{B})$ as input to decrypt π into k_u. Moreover, we rely on the failure to correctly recover k_u at the *time of use* if there had been any tampering with the binary since the *time of check*. It is possible to targetedly change a selected portion of a binary in memory, after it has been read to generate the decryption key. This, however, does not lead to a practically exploitable time-of-check time-of-use (TOCTOU) attack: An attacker is not able to discern which instruction actually failed during the manipulated binary's execution and for what exact reason. So (s)he can just resort to blind guessing attacks with the success probability of breaking the encryption scheme itself. The attack, therefore, does not scale since each single instruction needs to be reverse engineered again by guessing. The encryption scheme ensures that the same instruction at another memory location is not discernible for the attacker this way, either.

In our PoC implementation, we use the well-known CTR mode for random access to instructions during execution. We are aware of a weakness in CTR that may be exploited for a theoretical attack under rare circumstances. With considerable effort of brute-forcing opcodes and in combination with the mentioned tedious TOCTOU, an attacker could substitute single instructions. Even if successful, the usage of the PUF prevents the predictability of the keystream for other binaries and other devices. Therefore, this attack does not scale to multiple target machines. If even stricter security properties are desired, CTR readily may be exchanged for other random access modes in an own implementation. As pointed out in Sect. 4, CTR is only one of the options as a suitable crypto-primitive for an implementation.

6.2 Performance

The PoC shows that our enhanced architecture can be implemented with identical clock rate as the baseline processor. Therefore, the performance of SEPP is identical to the baseline system with the exception of the decryption. We therefore concentrate our discussion on the performance impact inflicted by decryption. Decryption latency and bandwidth potentially impact our architecture's performance twofold:

First, in a sequential stream of instructions, the processor would be stalled if the bandwidth of the decryption module would be smaller than the processor's bandwidth; we call this *execution latency*. As described in Sect. 5, our PoC demonstrates that a hardware-implementation is possible with a decryption-throughput high enough to completely prevent inflicting any execution latency. Second, upon each jump, the processor has to be stalled until the newly fetched instruction is decrypted, causing the so-called *warm-up latency* lat_w. Block cipher modes of operation, like the AES-CTR we used in our PoC, requires a couple of cycles to warm-up. The number of cycles, before the cipher yields the first block of output, depends on the implementation of the block cipher. Consequently, the only remaining occurring performance impact during execution is the warm-up latency at the beginning of basic blocks.

Calculating the Performance Penalty: Since the warm-up latency occurs at branching instructions, the overall performance penalty is dependent on the control flow of the program. We can calculate the runtime penalty by normalizing the number of clock cycles required to execute an encrypted program on the SEPP platform to the number of clock cycles required to execute an unencrypted program on the baseline platform:

$$runtime\ penalty = \frac{IC \cdot CPI + BIC \cdot lat_w}{IC \cdot CPI},$$

where IC denotes the total number of executed instructions of a program and CPI the average *clock cycles per instruction*. The values of IC and CPI are identical for SEPP and the baseline processor. Therefore, the overhead can be calculated by $BIC \cdot lat_w$ as the product of the number of branching instructions (BIC) and the warm-up latency in clock cycles (lat_w).

The AES implementation in our PoC system requires a warm-up latency of 13 clock cycles. Moreover, every processor has a latency induced by each memory access, for OR1200 and SEPP this is 5 cycles on average. The average memory access latency and CPI are rough estimations based on the OR1200's data sheet and system simulations. For a hypothetical program with 1 mio. instructions, 10% branching instructions and a CPI of 1.5, the runtime penalty calculates to 59% of SEPP compared to the baseline.

We validated this theoretical calculations by reducing the number of jump instructions in a actual binary from 14.3% to 5.3%. Compared to the baseline system, the performance penalty decreases from 84% to only 34% for our PoC measurements. This clearly demonstrates the impact of branch and jump

instructions on the execution time of encrypted programs. Appendix B contains details about the practical performance measurements which we conducted.

Remark: Recent advances in the design of hardware block ciphers promise a significant reduction of the warm-up latency. Since our architecture is not restricted to AES-CTR, but allows for arbitrary random access modes of suitable block ciphers, other ciphers can easily be substituted for AES. Low-latency block ciphers like PRINCE [6] can perform encryption in a single cycle at 14 to 15 times less area cost than AES-128. With an estimated average of 5 cycles per memory access, the performance penalty inflicted by the warm-up latency drops below the memory access latency. It needs to be determined at which length of the critical path the yielded throughput allows for uninterrupted execution. However, we argue, that such an implementation of the processor would come without any performance penalty whatsoever, compared to the baseline, but at the cost of a lower security level [21].

7 Related Work

Secure Code Execution. To our knowledge, there is no previous description of a system design that embeds security as deep into a chip as our architecture. This kind of architectures have been termed Isolated Execution Environment (IEE) [25]. Several recent works on IEEs [10,25,28,29,31] focus on confidentiality of processed user data and in addition, aim to minimize the side channel attack surface of processors. However, comparatively few papers address the confidentiality of application code which is the goal of our design. For these approaches, the assumed attacker capabilities vary substantially.

The *eXecute-Only Memory* (XOM) architecture [19], for example, considers main memory to be insecure but on-chip memory like caches to be secure. All data including code is encrypted when it leaves the cache and decrypted when it is brought back from main memory. It has been shown that this leads to substantial latency issues of memory accesses which is enhanced by Yang et al. [29]. In contrast, our approach considers all memory and caches as untrusted and thus stores only encrypted code.

Ascend [10] is a secure CPU architecture designed to obfuscate input and output signals on the processor's pins. The architecture has been extended into Stream-Ascend [31] to overcome Ascend's rather harsh limitations on the processors interactions with the outside world. Ascend's main goal is to obfuscate and minimize side channels due to access timings of off-chip data transfers. The capabilities of Ascend can be considered orthogonal to our architecture.

The *AEGIS* secure processor [27,28] is the first attempt to utilize the challenge-response behavior of a PUF for an IEE. Like the XOM processor, AEGIS encrypts only main memory using a similar OTP encryption scheme. The encryption keys are derived from the embedded PUF. Applications can switch the processor to different secure execution modes to match the current security demands. This is very flexible and improves performance compared to full

program encryption but requires careful consideration by the application's programmer, which makes porting existing software to AEGIS a non-trivial task. In addition, AEGIS requires extensive compiler and OS support, as well as modified hardware like a custom memory controller. Compared to AEGIS, our approach aims for a smaller TCB where code is only decrypted directly in the execution pipeline. As we show, we provide better compatibility with existing code.

OASIS is an instruction set extension for secure CPUs which provides an IEE for secure execution and remote attestation [25]. All cryptographic keys are bootstrapped from a PUF secret generated by an SRAM PUF [11]. Data confidentiality and integrity is established by encrypting program data with keys bound to the program code. However, unlike our approach, OASIS does not encrypt the code itself.

Intel SGX provides security enclaves for isolated and secure execution of programs [9]. To provide and manage enclaves, a large TCB is necessary, encompassing not only the processor logic itself. SGX assumes that everything on-chip is trusted, including caches and memory management. To provide confidentiality and integrity for data residing outside of the processor, it requires a complex Memory Encryption Engine. Thus, SGX does encrypt program code on transfer to the external memory, but it is decrypted transparently on access such that data and code is available in clear-text within the processor. The separation between enclaves, to prevent them from reading data from each other, is hardware-supported but managed completely by the security kernel in software. So the TCB is further enlarged by the necessity of a complex security kernel.

In summary, most presented approaches are data-centric and attempt to secure communication to and from the processor by introducing an additional encryption or obfuscation layer. They essentially localize security at the memory interface. Once an attacker gains access to CPU internal caches or pins, instructions are unencrypted and can be read out or modified.

In our architecture, we propose a deeper embedding into the execution pipeline where code remains encrypted in all memory and caches—even the instruction cache. Previous architectures come at the price of a relatively large TCB, including most of the processor and memory attachment, in some cases even large portions of software with parts of the operating system. This requires modified compilers, with significant customizations. Thus, the programming models of such concepts differ significantly from conventional ones.

Our design aims at minimizing such obstacles to the adoption of a secure IEE, so that it can be integrated much easier into existing environments.

Physical Unclonable Functions and Fuzzy Extractors. Storing encryption keys in memory inside or – even worse – outside of the CPU is a source of potential vulnerabilities, as attackers may succeed in extracting them, e.g., by the use of cold-boot attacks. In our design, we utilize the physical diversity of chip hardware to deduce a device-unique key that does not need to be stored in expensive secure non-volatile memory.

Physical unclonable functions (PUFs) evaluate manufacturing variations in integrated circuits to derive unique secrets inside a device to generate

cryptographic keys or authenticate a device in a challenge-response protocol [2,15]. Instead of storing secrets permanently, PUFs reveal their secret only during runtime. Popular PUF types for key generation are the SRAM PUF [11] and the Ring-Oscillator (RO) PUF [23].

The secrets derived from PUFs are noisy and affected by environmental conditions such that they require additional error correction in form of a so-called Fuzzy Extractor or Helper Data Key Extractor. Helper data is generated to map the random PUF responses to codewords of an Error-Correcting Code (ECC), thereby eliminating the variation in the PUF responses. Implementations of related approaches are based on the Code-Offset construction [7] , the Syndrome construction [22] or Differential Sequence Coding [13,14]. In this work, we use a Complementary Index-Based Syndrome coding (C-IBS) RO implementation [12] which is an extension of IBS [30]. The implementation contains a small Reed-Muller code with GMC soft-decision decoding [8] as ECC.

8 Conclusions and Future Work

In this paper, we presented a PUF-based secure code execution architecture to prevent reverse engineering of programs and to counter injection of malicious code into memory and cache to enhance the security of embedded devices. Deploying confidential code on our platform provides for additional application security and an enhanced protection of intellectual properties of the software developer. Novelties of this architecture are to embed a PUF-based decryption module, as deep as into the instruction fetch stage of the CPU's pipeline. This allows for a significantly smaller trusted computing base than previous designs of a secure processor. By employing a PUF we devise a new cryptographic protocol to allow binding of decryption to a target hardware instance. Encryption remains possible at a trusted development machine without access to the target. Despite these benefits, the architecture's minimal impact on the software development process simplifies application migration.

For our proof-of-concept implementation we used AES in CTR mode as a widely accepted mode of operation. Like all employed crypto-primitives, AES-CTR mode can be exchanged for a number of other block cipher modes suited for any individual security demands and addressing practical requirements. Depending on the acceptable area overhead, branching penalties may be removed almost completely by doubling the block cipher cores and compute two keystreams for both control flow branches in parallel. The flexible architecture we proposed allows for the customization of any implementation to meet the individual requirements of the specific use-case, for which our PoC implementation (SEPP) is one example. We leave it to future work to evaluate the differences of multiple incarnations of our architecture besides SEPP, regarding security impact, performance penalty, and chip-area requirement. Ultimately, this ought to result in an ASIC implementation for productive use.

Our SEPP showed, that direct and immediate placement of the PUF-driven instruction decryption, right into the instruction fetch stage, can be realized

and allows reasonable performance. Compared to related work evaluated on an FPGA, our SEPP has a similar overall performance penalty. At the same time, our architecture allows SEPP to reach the envisioned stricter security goals by moving the security deeper into the hardware, as proposed by Paul Kocher at his CRYPTO/CHES 2016 talk on the Future of Embedded Security. The reduced attack surface prevents the success of any kind of practical attack for reverse engineering and injecting code on memory and caches – even the instruction cache – which related secure architectures do not address in particular. SEPP's architecture achieves these properties with minimal impact on the development of software.

Appendices

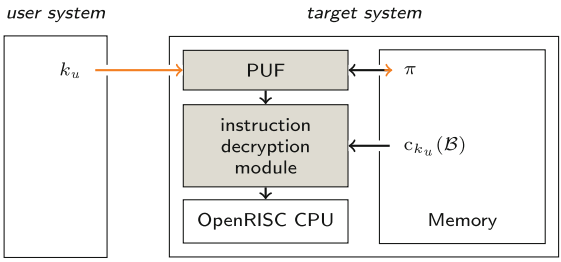


Fig. 4. SEPP architecture building blocks overview (orange arrows: key generation, black arrows: execution) (Color figure online)

A Deployment of Software Updates

Programs are initially flashed to a SEPP-powered System-on-a-Chip (SoC) at deployment time by the user via a secure connection to the device, e.g., a dedicated cable. To enable software updates in the field, an encrypted binary $c_{k_u}(\mathcal{B})$—initially deployed in a secure environment as stated above—may contain an *update function*. This update function needs to be able to take a new user-key $k_u{}'$ and a new encrypted binary $c_{k_u{}'}(\mathcal{B}')$ and feed it into the PUF-driven executable-image generation process. SEPP will return $\pi' = c_{k_p{}'}(k_u{}')$ to be packaged into a new image $(c_{k_u{}'}(\mathcal{B}'), \pi')$ for its current instance. A user sends $c_{k_u{}'}(\mathcal{B}')$ and $k_u{}'$ to this function; the encrypted $c_{k_u{}'}(\mathcal{B}')$ can be transmitted via an untrusted network connection, while $k_u{}'$ needs to be transmitted securely. To provide this secure channel between user and $\mathcal{B}$ running on the SEPP device, the update function already deployed in the trusted binary $\mathcal{B}$ must contain a suitable encryption scheme, e.g., RSA-based such as TLS or SSH. Due to the small key size, the secure channel for $k_u{}'$ requires only a low data rate.

The PUF-driven executable-image generation process of SEPP, generating $\pi = c_{k_p}(k_u)$, has to be deactivated physically after deployment of the binaries for *ultimate* security demands. This completely prohibits the generation of any new executable sequence of instructions for this instance of SEPP and therefore

completely prevents injections. For the over-the-air update, this security feature has to be deactivated as trade-off for an update path. The executable-image generation process of SEPP alternatively can only be triggered by a privileged instruction. This restricts its usage to the security kernel, however shifting the responsibility into software to decide what trusted code is allowed to generate new code.

B Practical Performance Measurements

Benchmarks and tests were conducted on our prototype implementation. We compare the test results with our prototype's base platform, the OR1200 CPU, running on the same FPGA board as SEPP's prototype implementation. In order to demonstrate the performance impact, we developed a number of custom tests with known parameters such as the number of branching instructions. These custom tests were all compiled without any compiler optimization in order to ensure deterministic outcomes. The results of our tests are summarized in Table 2.

Table 2. Results of custom tests executed on prototype implementation

		$\frac{2}{3}$ of jumps removed	manual loop unroll
number of branching instructions	unmodified	14.3%	9.8%
	modified	5.3%	3.8%
baseline system			
iterations per second	unmodified	909,091	138,122
	modified	1,063,830	183,824
speed-up factor		1.17	1.33
SEPP			
iterations per second	unmodified	495,050	85,690
	modified	793,651	157,233
speed-up factor		1.6	1.83
runtime penalty compared to baseline system	unmodified	1.84	1.61
	modified	1.34	1.17

By reducing the number of jump instructions from 14.3% to 5.3% through the removal of a function call within a loop, compared to the baseline system, the performance penalty decreases from 84% to 34%. This clearly demonstrates the impact of branch and jump instructions on the execution time of encrypted programs.

Loops generally introduce a substantial number of jumps into program execution. To verify this assumption, we manually unrolled a loop of a short example program. The number of branch instructions thereby decreased from 9.8% to 3.8%, reducing performance penalty from 61% to 17% for our custom test cases.

Furthermore, we performed CoreMark[4] benchmarks, to show the influence of unrolling loops and to enable comparison with future developments and other platforms. We conducted the benchmark both on our system in encrypted form as well as on the baseline architecture in unencrypted form. We compare the benchmark compiled with GCC's optimization level `-O3` with another version that is compiled with additional loop unrolling optimization, enabled with the flag `-funroll-all-loops`. Compilation with `-O3` results in a 48.8% penalty of SEPP, and enabling unrolling reduces this to 43.5%. While the baseline processor only speeds up by 9% with the additional `-funroll-all-loops`, SEPP's execution speed gains 21%. This confirms that the reduction of branching instruction benefits SEPP greatly. These benchmark comparisons prove the expected effects of our instruction-level decryption, specifically the warm-up latency on branching, for actual calculations.

Unfortunately, the comparison of our prototype with related implementations is not straightforward, as authors in this field use a variety of methods for performance assessment. The AEGIS developers used the SPEC2000 CPU[5] benchmark suite [27], which is not freely available. Lie et al. [19] provide a theoretical performance analysis for their XOM architecture [19]. The OASIS instruction set extension is not directly comparable to SEPP, as the authors only give absolute time overheads for specific platform operations [25].

The AEGIS developers state that degradation can be as high as 60%; while it mostly stays below 40%. XOM's slow down is less than 50% according to the authors' calculations. Given this comparison, we conclude that the runtime penalty of SEPP lies well within worst case, best case, and average for comparable systems. We expect a significant performance advantage of SEPP over the compared platforms when the described improvements are implemented in future work.

We are not able to compare SEPP to previous work in respect to FPGA resources overhead for the lack of information about properties of those approaches: XOM is only an architectural design without a hardware implementation [19]. OASIS was only simulated in software [25]. Although, an FPGA implementation of AEGIS was evaluated [28] no information about their FPGA resource requirements were given. Its resource overhead was only determined by an ASIC synthesis. The ASIC area overhead of AEGIS was given as being roughly 90% larger than their baseline. We consider this value not to be directly comparable to the roughly 68% Slices overhead of SEPP.

[4] http://www.eembc.org/coremark accessed on 27/02/2016.
[5] https://www.spec.org/cpu2000/, accessed on 27/02/2016.

References

1. Ahmed, S., Samsudin, K., Ramli, A.R., Rokhani, F.Z.: Effective implementation of AES-XTS on FPGA. In: 2011 IEEE Region 10 Conference (TENCON) (2011)
2. Armknecht, F., Maes, R., Sadeghi, A.R., Standaert, F.X., Wachsmann, C.: A formal foundation for the security features of physical functions. In: S&P. IEEE (2011)
3. Barrantes, E.G., Ackley, D.H., Forrest, S., Stefanović, D.: Randomized instruction set emulation. ACM Trans. Inf. Syst. Secur. **8**(1), 3–40 (2005)
4. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The SIMON and SPECK lightweight block ciphers. In: DAC (2015)
5. Bogdanov, A., Mendel, F., Regazzoni, F., Rijmen, V., Tischhauser, E.: ALE: AES-based lightweight authenticated encryption. In: Moriai, S. (ed.) FSE 2013. LNCS, vol. 8424, pp. 447–466. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43933-3_23
6. Borghoff, J., et al.: PRINCE – A low-latency block cipher for pervasive computing applications. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 208–225. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34961-4_14
7. Bösch, C., Guajardo, J., Sadeghi, A.-R., Shokrollahi, J., Tuyls, P.: Efficient helper data key extractor on FPGAs. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 181–197. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85053-3_12
8. Bossert, M.: Channel Coding for Telecommunications. Wiley, Hoboken (1999)
9. Costan, V., Devadas, S.: Intel SGX Explained. IACR **2016**(86), 1–118 (2016)
10. Fletcher, C.W., Dijk, M.V., Devadas, S.: A secure processor architecture for encrypted computation on untrusted programs. In: STC. ACM (2012)
11. Guajardo, J., Kumar, S.S., Schrijen, G.-J., Tuyls, P.: FPGA Intrinsic PUFs and Their Use for IP Protection. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 63–80. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74735-2_5
12. Hiller, M., Merli, D., Stumpf, F., Sigl, G.: Complementary IBS: application specific error correction for PUFs. In: HOST. IEEE (2012)
13. Hiller, M., Sigl, G.: Increasing the efficiency of syndrome coding for PUFs with helper data compression. In: DATE. ACM/IEEE (2014)
14. Hiller, M., Weiner, M., et al.: Breaking through fixed PUF block limitations with differential sequence coding and convolutional codes. In: TrustED. ACM (2013)
15. Katzenbeisser, S., Kocabaş, Ü., Rožić, V., Sadeghi, A.-R., Verbauwhede, I., Wachsmann, C.: PUFs: myth, fact or busted? A security evaluation of physically unclonable functions (PUFs) cast in silicon. In: Prouff, E., Schaumont, P. (eds.) CHES 2012. LNCS, vol. 7428, pp. 283–301. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33027-8_17
16. Krawczyk, H.: Cryptographic extraction and key derivation: the HKDF scheme. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 631–648. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14623-7_34
17. Kömmerling, O., Kuhn, M.G.: Design principles for tamper-resistant smartcard processors. In: Proceedings of the 1st Workshop on Smartcard Technology (1999)
18. Langner, R.: To kill a centrifuge - a technical analysis of what stuxnet's creators tried to achieve. Technical report. The Langner Group, November 2013
19. Lie, D., Thekkath, C., Mitchell, M., et al.: Architectural support for copy and tamper resistant software. SIGOPS Oper. Syst. Rev. **34**(5), 168–177 (2000)

20. Liskov, M., Rivest, R.L., Wagner, D.: Tweakable block ciphers. J. Cryptol. **24**(3), 588–613 (2010)
21. Maene, P., Verbauwhede, I.: Single-cycle implementations of block ciphers. In: Güneysu, T., Leander, G., Moradi, A. (eds.) LightSec 2015. LNCS, vol. 9542, pp. 131–147. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29078-2_8
22. Maes, R., Van Herrewege, A., Verbauwhede, I.: PUFKY: a fully functional PUF-based cryptographic key generator. In: CHES (2012)
23. Maiti, A., Schaumont, P.: Improved ring oscillator PUF: an FPGA-friendly secure primitive. J. Cryptol. **24**(2), 375–397 (2011)
24. One, A.: Smashing the stack for fun and profit. Phrack **7**(49) (1996)
25. Owusu, E., Guajardo, J., et al.: OASIS: on achieving a sanctuary for integrity and secrecy on untrusted platforms. In: SIGSAC. ACM (2013)
26. Simpson, E., Schaumont, P.: Offline hardware/software authentication for reconfigurable platforms. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 311–323. Springer, Heidelberg (2006). https://doi.org/10.1007/11894063_25
27. Suh, E.G., Clarke, D., Gassend, B., van Dijk, M., Devadas, S.: AEGIS: architecture for tamper-evident and tamper-resistant processing. In: ICS. ACM (2003)
28. Suh, E.G., et al.: Design and implementation of the AEGIS single-chip secure processor using PUFs. SIGARCH Comput. Archit. News **33**(2), 25–36 (2005)
29. Yang, J., Zhang, Y., Gao, L.: Fast secure processor for inhibiting software piracy and tampering. In: International Symposium on Microarchitecture. IEEE/ACM (2003)
30. Yu, M.D., Devadas, S.: Secure and robust error correction for physical unclonable functions. IEEE Des.Test Comput. **27**(1), 48–65 (2010)
31. Yu, X., Fletcher, C.W., et al.: Generalized external interaction with tamper-resistant hardware with bounded information leakage. In: CCSW. ACM (2013)