Secure Execution Architecture based on PUF-driven Instruction Level Code Encryption

Stephan Kleber[1], Florian Unterstein[1], Matthias Matousek[1], Frank Kargl[1], Frank Slomka[2], and Matthias Hiller[3]

[1] Institute of Distributed Systems, Ulm University, Germany, `firstname.lastname@uni-ulm.de`
[2] Institute of Embedded Systems/Real-Time Systems, Ulm University, Germany, `frank.slomka@uni-ulm.de`
[3] Institute for Security in Information Technology, Technische Universität München, Germany, `matthias.hiller@tum.de`

Abstract. A persistent problem with program execution, despite numerous mitigation attempts, is its inherent vulnerability to the injection of malicious code. Equally unsolved is the susceptibility of firmware to reverse engineering, which undermines the manufacturer's code confidentiality. We propose an approach that solves both kinds of security problems employing instruction-level code encryption combined with the use of a physical unclonable function (PUF). Our novel *Secure Execution PUF-based Processor* (SEPP) architecture is designed to minimize the attack surface, as well as performance impact, and requires no significant changes to the development process. This is possible based on a tight integration of a PUF directly into the processor's instruction pipeline. Furthermore, cloud scenarios and distributed embedded systems alike inherently depend on remote execution; our approach supports this, as the secure execution environment needs not to be locally available at the developers site. We implemented an FPGA-based prototype based on the OpenRISC Reference Platform. To assess our results, we performed a security analysis of the processor and evaluated the performance impact of the encryption. We show that the attack surface is significantly reduced compared to previous approaches while the performance penalty is at a reasonable factor of about 1.5.

1 Introduction

Today, many embedded systems are no longer stand-alone but are connected by field busses, like CAN or Flexray in the automotive domain and Profibus in production environments. A growing extent of such systems is also connected by standard Internet technologies; and in this context the term cyber physical system is used. This immediately raises the issue of security and the possibility of remote attacks. One big challenge is the secure execution of programs.

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) in the Windows RPC handler to infect remote machines [17]. Since the infamous phrack article "Smashing The Stack For Fun And Profit" [25] by Aleph One in 1996, which described simple stack buffer overflows, many additional detection and prevention techniques like stack canaries or non-executable stacks have been proposed and soon thereafter been circumvented by more sophisticated attack techniques. Now, more than 15 years later, it is still an open security challenge to effectively prevent injection of unauthorized code into an execution environment. In the case of cyber physical systems, insecure execution of programs may result in successful hacker attacks and potential danger to life and expensive equipment.

Goals of a secure and isolated execution environment are (1) to protect against code injection to prevent malicious actions inside the environment *(code injection)* and (2) to prevent genuine code from getting extracted out of its execution environment to prevent reverse engineering *(code confidentiality)*. In this paper we present *Secure Execution PUF-based Processor* (SEPP), a novel processor architecture which allows secure execution of encrypted programs while encrypted program images can only be generated with the help of the target processor instance itself. Code that is not properly encrypted will not execute and thus an attacker will not be able to produce and remotely inject code. At the same time, the architecture will effectively prevent extraction and reverse engineering of programs stored in such an embedded system. While we focus on cyber physical systems in this paper, there are many additional application areas, e. g. in cloud computing. Based on the presented architecture, future work is intended to allow the integration of secure remote code deployment for cloud computing.

The contribution of this paper is an architecture for a secure execution environment based on a processor providing runtime decryption of single instructions directly in the execution pipeline. A central security feature of this instruction level encryption is a strict hardware binding of code through utilization of a PUF. It cryptographically restricts executable code to the single instance of our secure processor it was specifically compiled for. Thereby injection of malicious code on the one hand and extraction of code for reverse engineering of deployed applications on the other is prevented, as long as developers keep their cryptographic development keys secret. The feasibility of the architecture is shown by an implementation and its theoretical and practical evaluation on a Xilinx Spartan-6 FPGA based on the OpenRISC architecture.

2 Related Work

2.1 Secure Code Execution

To achieve code injection mitigation and code confidentiality, it is necessary to maintain control over the execution environment, even if it is physically accessible to an adversary. Previous attempts to solve similar problems have been described using the term isolated execution environment (IEE) [26]. Several recent works on IEEs [36, 9, 26, 29, 34] have shown that it is possible to keep data confidential and to minimize the side channel attack surface of processors. However, few papers concerned the confidentiality of application code.

Binding software to a physical execution environment to build a trusted computing base, tries to solve the problem on a fundamental level involving the underlying hardware as anchor of trust. This has been the objective of several initiatives [31, 24, 2] for the last decade. However, the adoption of those methods has been scarce presumably for concerns over poor performance, complex deployment and unsatisfactory protection against local attacks [26].

Today's computing systems include many extensions to prevent code injection attacks. One popular method is the insertion of a so-called *canary* variable around dynamic memory to detect an attack [8, 14]. Other approaches use randomization to prevent attackers from making assumptions about their victim. *Address space layout randomization* (ASLR) randomizes the location of address segments like stack, heap or libraries in a process' address space. It is available in most modern operating systems, including Windows, Linux and Android. In contrast *instruction set randomization* (ISR) randomizes machine instructions [16]. Code injected without knowledge of the current instruction set will most likely cause a runtime exception [4]. A hardware-supported countermeasure is *executable space protection*. It allows to mark certain memory areas as non-executable and prevents the processor from executing them. This feature is known as *NX-bit* on AMD processors and *XD-bit* on Intel processors.

The *execute-only memory* (XOM) architecture [19] considers main memory to be insecure and assumes on-chip memory like caches to be secure. In XOM, all data is encrypted when it leaves the cache and decrypted when it is brought back from main memory. This leads to significant latency issues when accessing memory [34]. Yang et al. address this problem by using an algorithm similar to one-time pad (OTP) encryption [34]. Our approach does not require the assumption that main memory or even caches are secure.

The *AEGIS* secure processor [28, 29] is the first attempt to utilize the challenge-response behavior of a PUF. 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. AEGIS also requires extensive compiler and OS support, as well as modified hardware like a custom memory controller. Our approach aims for a smaller trusted computing base (TCB) and better compatibility with existing code.

OASIS is an instruction set extension for secure CPUs which provides an isolated execution environment for secure execution and remote attestation [26]. All cryptographic keys are bootstrapped from a PUF secret generated by an SRAM PUF. 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. To protect the actual execution, the processor uses a Cache-as-RAM mode using on-die cache as general purpose memory to perform computations.

Ascend [9] 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 [36] to overcome the 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 are orthogonal to our architecture.

All 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 locate security at the memory interface. Once an attacker gains access to caches or pins, instructions are unencrypted and can be read out or modified. We aim for a deeper embedding where code remains encrypted in memory and caches and gets decrypted just within the execution pipeline. 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 specialized compilers, with significant changes compared to standard compilers for the base line architecture of each respective approach. The programming models of those concepts also differ significantly from conventional ones. Our design aims at minimizing such adverse effects of a secure IEE.

2.2 Physical Unclonable Functions

Storing encryption keys in memory inside or 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. *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 [3, 15]. Instead of storing secrets permanently, PUFs reveal their secret only during runtime. Popular PUF types for key generation are the SRAM PUF [10] and the Ring-Oscillator (RO) PUF [30, 23]. The secrets derived from PUFs are noisy and affected by environmental conditions such that they require additional error correction. 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 [6, 21, 18], the Syndrome construction [22] or Differential Sequence Coding [13, 12]. In this work, we use a Complementary Index-Based Syndrome coding (C-IBS) RO implementation [11] which is an extension of IBS [35]. The implementation contains a small Reed-Muller code with GMC soft-decision decoding [7] as ECC.

3 Preliminaries

3.1 Encryption ciphers

There are two major types of encryption ciphers: stream ciphers and block ciphers. Although intuitively the right choice, stream ciphers have several undesirable properties in our use case: They need a warm-up phase of several hundred cycles and it is not possible to randomly jump from one position in the stream to another. This renders them unusable for instruction stream encryption because random access is mandatory.

Block ciphers operate on a block of plaintext with a fixed size and encrypt it using a symmetric key. Because the input length is limited to a fixed size of bits, several modes of operation exist to encrypt longer plaintexts. In counter (CTR) mode, a nonce is combined with a counter value and encrypted as shown in Fig. 1. The result is used as encryption pad and XORed with the plaintext. The

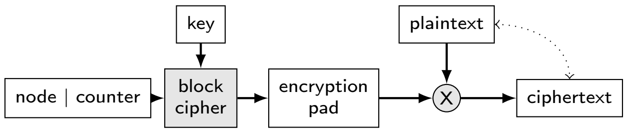


Fig. 1. Block cipher in CTR mode of operation

XOR operation can be calculated in hardware in less than one clock cycle and induces no execution delay. For the next block, the nonce stays the same and the counter is incremented. To decrypt a message, the ciphertext is again XORed with the encryption pad. Such a block cipher in CTR mode has the same benefits as regular stream ciphers and is in addition capable of random access as pads for any block can be calculated directly.

3.2 Adversary Model

There are two distinct addressed adversary models:

For the **code confidentiality** scenario (e.g., to protect intellectual property of programs in embedded systems), we assume that the attacker has physical access to the processor and its peripheral connections, including network components as well as low-level memory bus lines. The attacker tries to learn parts of the application code by reading out memory or registers. Programs are flashed to embedded micro controllers at production time by the manufacturer using a secure connection to the device.

The provider is certified to provide the expected processor architecture. For this certification we assume the hardware manufacturer to be a trusted third party. The users system is assumed to be secure and a cryptographically secured channel between user system and remote processor can be established. We do not consider all details of this connection within this work.

For the **malicious code injection** scenario, we assume an attacker that may or may not have physical access to the processor, but has the ability to place arbitrary data into the processors main memory. We deliberately do not limit the access to certain portions of memory. Vulnerabilities introduced accidentally by programming errors like buffer overflows, likewise are conceptually not limited to certain memory areas and we therefore address the mitigation of malicious code injection on this general level.

Beyond those two primary scenarios, attacks like denial-of-service (DoS), e. g. injecting random invalid instructions, and hardware side-channels are not specifically considered within our approach. Despite this, we like to note that complementary research exists, attempting to solve issues of hardware side-channels. Ascend [9], for example, is specifically designed to hamper side-channel attackers by obfuscating memory access patterns, power consumption and temperature analyses. It does so by restricting instances of memory access to fixed public points in time, at which a number of memory operations are performed, even if none are necessary for the actual task of the arithmetic logic unit (ALU). The ALU on the other hand, fires all possible data paths on all components at each cycle so that the actual instruction can not be determined by power or thermal analyses of chip regions. We concur with the authors of XOM, AEGIS, OASIS, and Ascend in assuming that there exists a variety of methods to prevent or impede hardware tampering, e. g. probing or fault-injection. Therefore we consider the chip itself a tamper-proof packaged piece of hardware and attacks of this kind are addressed only implicitly.

4 Secure Execution PUF-based Processor Architecture

In embedded systems, execution typically happens detached from development. To address our envisioned use cases, we separate the SEPP-architecture into a development machine and the execution environment in the embedded system. The execution environment, where code is securely executed, we call *target system*. The development machine we call *user system*. On the user system, programs are generated by the user to be deployed later on the target system.

As shown in Fig. 3, generation of a program image at the user system requires the compilation and encryption of the code. The encrypted binary is then packaged as image to be transferred to the target machine. There, the image is cryptographically bound to the physical instance of the one processor the binary will solely be executable on in the future. This binding also prevents malicious code injection.

For our prototype, we chose OpenRISC as our platform, as it is a popular open-source RISC architecture. This architecture implies that a single instruction is of fixed 32-bits-length. This is convenient for the proposed instruction level encryption where the smallest encryption unit therefore is of the fixed length of one instruction. We extended the OpenRISC Reference Platform (ORPSoC) by a PUF module and an instruction decryption module.

The roles of these components of the architecture during program generation and program execution are explained in the following sections.

4.1 Program Generation

After a user has compiled code into a program binary on the trusted user system, he needs to encrypt the binary. The choice of a suitable encryption scheme has to take into account the requirements of the program flow. Program flow generally must allow for jumps (branches) and loops, so random access within the instruction stream is mandatory. The program flow typically is divided into basic blocks. A basic block is a sequence of instructions with exactly one entry point and one exit point, and no branches in between. Thus, it is possible to encrypt the program code on a basic block level, although it is the individual instructions that are decrypted before being executed. Encryption of basic blocks, using a symmetric cipher in CTR mode, enables random access to support branches. We use virtual addresses to identify code locations. Virtual addressing is convenient since it is static no matter where the program actually will reside. This requires a memory management unit which might not always be available, especially on low-cost systems where, however, alternative solutions are possible.

For this symmetric encryption, a key k_u is chosen by the user. Each basic block is encrypted using k_u with the following CTR mode parameters: The nonce is set to the virtual address of the beginning of the basic block and the counter is set to zero. A basic block may consist of multiple CTR mode encryption blocks. The counter is incremented for each encryption block while the nonce stays the same until the end of the current basic block is reached (Fig. 3).

With the start of the next basic block, the nonce is again set to the new basic block's starting address and the counter is cleared. This scheme is applied until the whole program is encrypted.

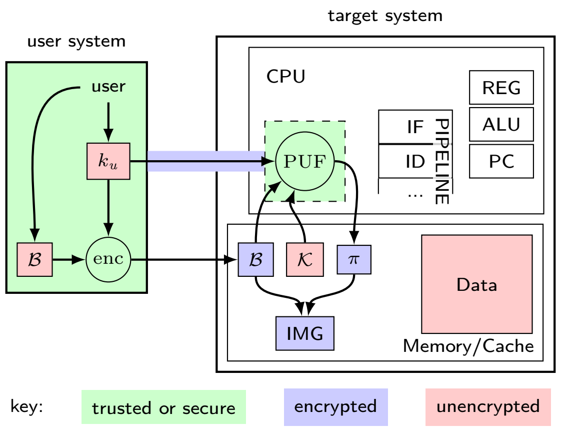


Fig. 2. SEPP architecture: **compile mode**

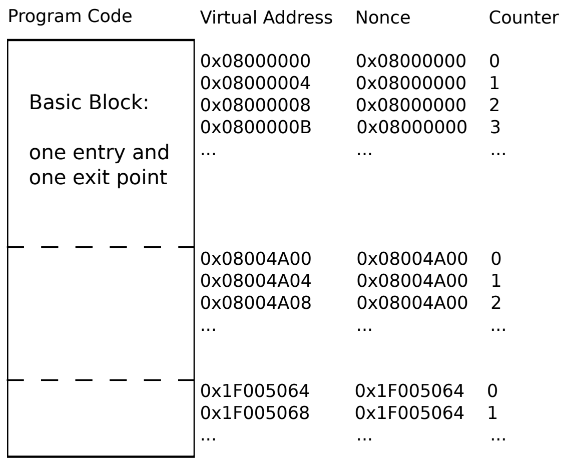


Fig. 3. Program encryption scheme

The program encryption itself is not critical for runtime performance and requires no hardware support. It may be implemented within the compiler or as standalone tool, processing the compiled binary in software.

To finally bind a program binary $\mathcal{B}$ to a hardware instance, inherent properties of the PUF are utilized. This part of the process of program generation is shown in Fig. 3. The encrypted binary $\mathrm{enc}_{k_u}(\mathcal{B})$ and k_u are thereto transmitted to the target system. The binary $\mathrm{enc}_{k_u}(\mathcal{B})$ may be public, but k_u must remain confidential, thus it has to be transmitted securely. For embedded systems, this transmission can be conducted before deployment over a dedicated data cable. The PUF is then used to generate a cryptographic key k_p, which is bound to the encrypted binary and – by the properties of the PUF itself – to the processor instance. In addition, the security kernel $\mathcal{K}$, i. e. all other necessary target-system software parts critical for security, is included in the computation. For this an HMAC is used:

$$c = \mathrm{HMAC}(\mathcal{K}, \mathrm{enc}_{k_u}(\mathcal{B})) \tag{1}$$

The challenge c is then used as input to the PUF, with k_p as the corresponding response.

The choice of an HMAC over a simple hash is not due to the security kernel $\mathcal{K}$ being a key to be incorporated, but to prevent length-extension attacks on the challenge c. LE attacks are only of concern when choosing a specific hash function, that follows the Merkle-Damgård structure, like MD5, SHA-1, and SHA-2. That said, HMAC is chosen for its property to be a MAC, i. e. a key-dependent hash function. Another way of viewing HMAC in this context is as a "symmetric digital signature scheme".

To protect the user key k_u, required for program-execution, it is encrypted using k_p and can then be stored publicly, together with the encrypted binary, to form the program image. This public representation of k_u we call $\pi = \mathrm{enc}_{k_p}(k_u)$.

This scheme has the advantage that the user does not require access to the target hardware itself to prepare a binary to be executed on it while retaining the desired security properties.

Constants such as data parameter or string sections of a binary are not to be encrypted in SEPP at this point. However, some embedded device vendors may rely on the confidentiality of input parameters of their algorithms rather than the secrecy of the algorithms themselves. We did not specifically address this issue, although we deem it feasible to let the secure program run any decryption algorithm on any encrypted data, whether residing in the binary itself or as external data. Necessary key material can be derived from the encrypted program itself without any hardware support.

4.2 Program Decryption and Execution

To minimize the parts of the processor required to be trusted, the decryption module is included in the instruction fetch stage of the processor's pipeline. Encrypted instructions enter the stage and are decrypted immediately before proceeding to the instruction decode stage. If the program execution starts, or an instruction jump is detected, the nonce is set to the current value of the program counter (PC) and the CTR mode counter is set to zero. This way, the CTR mode's encryption parameters are enforced to match the basic block decryption. The module pre-computes encryption pads by incrementing the counter value. The end of a basic block is reached when either a jump occurs or the execution runs into the next basic block. In either case, the processor detects the new block and restarts the decryption by resetting the counter. Some instructions take more than one processor cycle to execute. Thus, a first-in, first-out (FIFO) buffer is added to store the produced encryption pads until they are needed by the processor.

A simplified schematic view of the decryption module embedded in the instruction fetch (IF) stage is shown in Fig. 4. There are four main functional blocks. The *nonce and counter generator* prepares the input to the block cipher. It is also responsible for detecting jumps and for rewinding the counter when the FIFO is filled. Since an encryption pad block can be larger than one instruction (here: factor four), an additional multiplexer following the FIFO is necessary. It

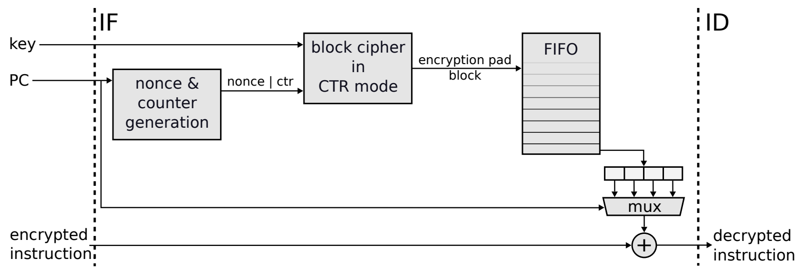


Fig. 4. Simplified structure of the decryption module

selects the correct part of the encryption pad block depending on the current PC. This encryption pad is XORed with the incoming encrypted instruction and the decrypted instruction is forwarded to the instruction decode stage. To simplify the schematic, all control logic has been omitted. The implementation also has to take care of resetting the encryption, flushing the FIFO and stalling the processor whenever necessary.

4.3 The Role of the PUF

The target system's memory might be attacked physically. Nevertheless, no expensive secure non-volatile memory is necessary, due to the program image being public. To remain self-contained, the correct k_u must to be recovered for execution only from the encrypted π inside the program image and k_p. Just the same processor instance can generate the right k_p to restore $k_u = \text{dec}_{k_p}(\pi)$.

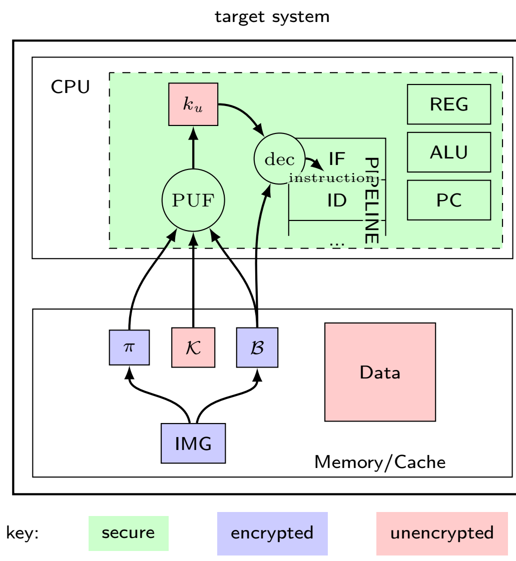


Fig. 5. SEPP architecture: **execute mode**

As depicted in Fig. 5, k_p has to be derived from the target system's PUF: $k_p = \text{PUF}(c)$ with the challenge c incorporating the security kernel and the program itself as described in Eq. 1. Sec. 4.1. Thus, c is unique for every program and infeasible to forge. Finding another binary that produces the same challenge is equally difficult as finding a collision in the hash function used for the HMAC. The HMAC breaks the direct link between user inputs and k_p which could otherwise be exploited. Key k_u needs to be present in the target system's instruction fetch (IF) logic during execution. This is necessary for the on-the-fly decryption of the instructions of a specific binary. However, k_u is not permanently stored or installed there. It has to be recovered from the publicly known π each time a binary is loaded for execution, to prevent any possibility of exposure. The target system does not store and requires no program specific private values except during execution.

The RO PUF implementation we utilized for our approach does not provide challenge-response behavior by itself. Instead, this PUF 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 (see Sec. 2.2) takes the PUF output and creates helper data to reliably recreate the fixed response. The corresponding helper data can be stored in hardware memory since an attacker gains no advantage from it. We call this reliable, embedded secret *PUF secret*, denoted s_p.

We now construct a challenge-response wrapper around the device specific s_p. Challenge-response behavior is required for the generation of cryptographic keys which are not only bound to the device but also to the program binary. To reach the desired challenge-response behavior, PUF and fuzzy extractor are integrated as shown in Fig. 6. The PUF module consists of the PUF with the challenge-

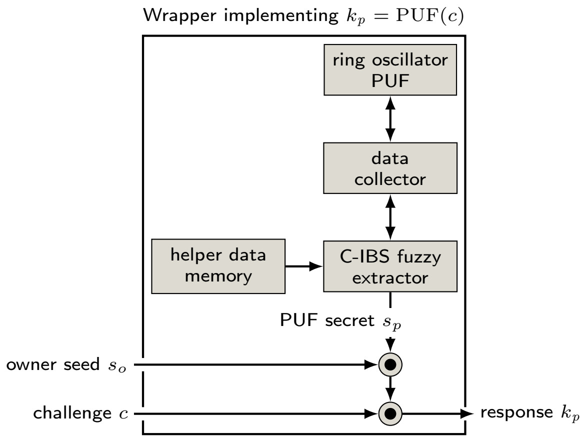


Fig. 6. PUF with challenge-response wrapper

response wrapper implemented by an HMAC (⊙). It is used as an alternative to a challenge-response PUF, so $k_p = \mathrm{PUF}(c) = \mathrm{HMAC}(s_p, c)$. For k_p to remain confidential, there is no interface to access the outputs of the PUF or the fuzzy extractor. To prevent an adversary from acquiring information about k_p when knowing k_u and π, it is important to use an HMAC.

The PUF challenge-response wrapper circuit is used in this context much like a keyed hash function. The key is the PUF secret k_p, combined with the owner seed s_o. The owner seed is an additional measure to restrict knowledge of third parties and is explained in the following section. The challenge is the encrypted binary $\mathrm{enc}_{k_u}(\mathcal{B})$ respectively a simple hash thereof. The response is k_p, which in turn is used as key for the encryption of $\pi = \mathrm{enc}_{k_p}(k_u)$, respectively decryption $\mathrm{dec}_{k_p}(\pi) = k_u$.

To prevent known-plaintext-attacks on any of the inputs (seed, PUF secret, challenge) and possibly the output of the encryption, π, a simple XOR is not sufficient. It further is insufficient to simply hash the resulting value, since length extension attacks might remain possible, due to the nature of the wide-spread Merkle-Damgård construction for hashes like MD5, SHA-1, or SHA-2. This is not primarily due to the fact, that a confidential key is needed in the input, but that an HMAC allows for the combination of multiple values, regardless of their confidentiality or interpretation as "key", without opening up to length extension attacks. Those attacks would endanger the confidentiality of either of the input values of a simple hash function. Therefore we use an HMAC here as well.

4.4 Preventing Leakage of Manufacturer Data

The PUF's challenge does not only consist of hashes of the binary and the security kernel, but also a device owner seed, combined with the PUF-created secret. This is necessary because part of the helper data has to be chosen by a party. The consequence is that the hardware manufacturer (HWM) can gain knowledge of the PUF secret. We tolerate this because we have to trust the HWM to build a PUF which is not manipulated or biased in any way that undermines security in the first place. Nevertheless, we want to prevent any party from being able to decrypt another users' confidential code. Notwithstanding the adversary model, the HWM's knowledge of s_p could leak to an attacker who has access to the encrypted program binary. The attacker can now compute the corresponding challenge (Eq. 1) and consequentially can compute k_p for every given encrypted program. Finally, the attacker can decrypt k_u and consequently the user's program.

To prevent this, an owner seed s_o is added. This seed is static and chosen by the device owner. Including this seed, none of the three parties involved knows the full challenge-response wrapper's input and can foresee its result: π, $\mathrm{enc}_{k_u}(\mathcal{B})$ and $\mathcal{K}$ are public; the user knows k_u and $\mathcal{B}$; the device owner knows s_o; the HWM may gain knowledge of s_p. Note that the PUF response is different on different devices, so s_p is unique to a device. The seed is encrypted with the device's public key so it can safely be stored outside the PUF module. Before the PUF can be evaluated for the first time, the seed is decrypted and saved in the PUF module. This gives the following equation for k_p, resolving the substitution of a challenge-response PUF:

$$k_p = \mathrm{HMAC}(c, \mathrm{HMAC}(s_o, s_p)) \tag{2}$$

The owner seed is set by a write instruction to a dedicated non-volatile register of the processor. There is no read operation to this register besides a direct line to the PUF-module. There is no possibility to extract this value without extremely sophisticated hardware probing which the adversary model excludes.

5 Prototype Implementation

To demonstrate the feasibility of our architecture, we developed a prototype capable of creating and executing encrypted standalone program images. The prototype is based on an OR1200 processor. The OR1200 is an implementation of the OpenRISC OR1000 architecture, an open-source RISC architecture with a 32 bit wide instruction set and a five stage, single issue pipeline. The full specification of our system is shown in Tab. 1. This baseline system was enhanced with two major modules: the PUF module and the instruction decryption module. We implemented our design on a Xilinx Spartan-6 LX45 FPGA.

The PUF module encapsulates the RO PUF itself with the C-IBS fuzzy extractor and a low area AES core in ECB mode for the encryption and decryption of k_u. The CPU interfaces the module over a dedicated class of special purpose registers. Those include configuration and status registers as well as the input of the PUF challenge and the write-only owner seed register. 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. 4 it evaluates the PC to detect branches and sets the nonce and counter accordingly. The program flow from one basic block to the other with no branch is marked by a custom instruction. Another custom instruction enables and disables the decryption to maintain compatibility with unencrypted programs. It also allows for mixed applications where non-critical parts might run faster in unencrypted mode and only specific parts of the code run in the secure encrypted mode. The AES decryption core should provide a throughput high enough to allow sequential code to be decrypted and executed without stalling. In order to minimize the area cost, we used four 13-cycle AES cores in parallel which are limited to decryption. They provide 16 decryption pads every 13 cycles which is enough for uninterrupted execution considering that in the worst case one instruction per cycle enters the pipeline. Thus, stalling the processor is only necessary until the first encryption pad for a basic block is computed.

The Universal Bootloader *Das U-Boot* (or U-Boot)[4] is used as software platform. It was modified to implement the functionality of the target system security

[4] `http://www.denx.de/wiki/U-Boot`, accessed on 22/02/2014

	OR1200
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

Table 1. Configuration of the prototype system

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/private key cryptography is used by U-Boot to establish a secure communication channel between the user system and the target system. The HMAC calculation for the PUF challenge is also implemented as part of our security kernel.

Program encryption is independent from the hardware prototype and is handled on the user system. In our case this 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 prototype system over an Ethernet connection. There π is generated and replaces the k_u embedded in the image. This image can now securely be made public as it only contains encrypted code and the public π. It is tied to this exact hardware instance and only this processor with its unique PUF is able to execute it.

6 Evaluation

6.1 Security

Through the use of a PUF, there is no unsecured key material that needs to be stored persistently. Nevertheless, to confirm the proposed security properties, considerations of the two adversary models (Sec. 3.2) *code confidentiality* and *injection prevention* need to be done.

Code Confidentiality To achieve code confidentiality, encryption of the binary image is applied. Conclusively the encryption and its necessary key material determines the security properties. The key necessary to decrypt the binary is k_u. Knowing a specific π and having available the encrypted binary and $\mathcal{K}$ to calculate the HMAC thereof, k_u can be recovered using the correct PUF instance. Only k_u and k_p need to remain confidential. k_p is expected to never leave the PUF module, whereas k_u in contrast originates externally. We therefore argue, that the confidentiality of a program equals the confidentiality of k_u outside of the processor's instruction decode module.

The single process where k_u is required outside of the processor is for the user to encrypt the binary after creation. Provided the adversary model and scope of our approach, the only possible 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. We assume that this can be ensured during deployment. The user machine itself must be secured and trusted. Those requirements are out of scope of our approach, but we are confident that they can be addressed by known means.

Injection Prevention The second scenario is prevention of code injection, which is accomplished by executing only correctly encrypted code. The security thereof is based on the confidentiality of k_p. The aforementioned decryption process of code requires the recovery of k_u, which in turn requires k_p. Provided k_p can not be extracted from the hardware, it only can be generated by the correct PUF instance. Blocking the key generation data path in the processor – most consequently by a hard-wired switch – the generation of a new valid π for any program can be prevented. Thereby no new validly encrypted code can be generated. Therefore injection of any valid instruction is as hard as finding a collision in the output of the block cipher generating k_u. In doing so, the attacker can not manipulate any of the block cipher's inputs directly except π itself.

Because of the per-basic-block encryption, the architecture is theoretically still vulnerable to control flow manipulation by attacks like return oriented programming (ROP). But as the counter value and nonce are controlled by hardware and cannot be influenced by the attacker, the most damage an attacker can cause are jumps to the beginnings of basic blocks. A jump to any other address will set wrong nonce and counter values and fail to decrypt the code at that location. This severely limits an attacker's ability to construct useful ROP gadgets. Due to the high probability that executing random bytes results in an exception [4], we argue that an attacker has no realistic way of injecting code. This is true as long as k_p is not extracted directly from the system by usage of very strong hardware attacks.

In general, SEPP is susceptible to a time-of-check time-of-use (TOCTOU) attack, which however is not practically exploitable. SEPP does not use any signatures of executable binaries. Instead we rely on the failure to correctly decrypt k_u at the *time of use* if there had been any tampering with the binary since the *time of check*. This likely failure of correct decryption results from the fact that the binary itself is used as part of the PUF-input providing the decryption key for π. But it is possible to targetedly change a selected portion of a binary in memory, after it has been read to generate the decryption key. Typically this change would remain dormant until the control flow reaches the manipulated operation. Only then the decryption of the binary at this position will result in some arbitrary value, most likely not being a valid instruction to SEPP. Even if it results in a valid instruction, it is likely that a subsequent instruction will fail since it depends on the outcome of this instruction. However, in general, the attacker is not able to discern which instruction actually failed during the manipulated binary's execution and for what exact reason. So he cannot even tell whether he successfully manipulated the last instruction executed or any of the previous ones. So, randomly guessing a value that correctly decrypts into a valid instruction is as difficult as breaking the encryption scheme. SEPP's usage of CTR-mode has a known weakness of this kind we will discuss later on to show that this is no actually feasible attack. Barrantes et al. [4] established that this kind of code encryption, used as integrity check of a binary, is a sufficiently large obstacle for an attacker under most practical circumstances. We especially emphasize this as a very good trade-of between security, practicality, overhead, and design decisions, since adding a signature to the binary would not automatically mitigate the TOCTOU attack. On the contrary, the attack would prevail and only increase the complexity of the design without any gain in security. Only repeated verifications of the assumed signature during each IF-cycle might solve this, but at the cost of a severe performance penalty, due to the memory access and signature verification operation. This remains true for every static code manipulation even if the PUF itself is involved in the assumed integrity check. Signature checking during each instruction-fetch-cycle might mitigate this attack completely, but at the cost of a severe performance penalty, due to the frequent memory access and signature verification operations. For our goal to protect against dynamic code injection in particular, the failure of decryption in the instruction fetch phase of the execution in practice is superior to any additional signature, when taking performance into account. Static code manipulation in dormant memory should be impossible in the first place using SEPP. This means that currently without dedicated signature to be checked is no easier for an attacker to replace portions of the encrypted binary code than with a signature.

There is a remote possibility for an attack due to the usage of the CTR-mode in the decryption phase. This attack exploits the fact that knowing the clear-text $\mathcal{B}^i$ at a position i in the binary $\mathcal{B}$ allows to replace this value by a freely chosen one, under certain circumstances. Knowing opcodes of the processor, those can targetedly be replaced by another known opcode in an encrypted binary but only during a successful TOCTOU-attack as explained above. Assume an attacker has gained knowledge of some opcodes and at least one exact usage position i of such an opcode in a certain binary. The attacker wants to exchange the binary value of this position $\mathcal{B}^i$ for an arbitrary number of i's by a value E^i. Therefore the attacker chooses values I^i to be injected so that $I^i = \mathcal{B}^i \oplus E^i$. At the processors instruction-fetch-logic, when XORing the decryption pad to the manipulated portion of the binary during execution, the following is decrypted:

$$(\mathcal{B}^i \oplus E^i) \oplus \mathcal{B}^i \oplus \mathrm{CTR}^i(k_u) = E^i \oplus \mathrm{CTR}^i(k_u)$$

Therefore the processor will execute E^i instead of $\mathcal{B}^i$. Another variation of this attack does not even require to know an opcode but only a single bit value of clear-text and the position for this to be injected purposefully. This single arbitrary bit can be flipped using the process outlined above. This is only helpful for DoS- or ROP-attacks. We excluded DoS for reasons discussed in the adversary model section. And since, as we argued earlier in the current section, SEPP only allows for a severely limited form of ROP-attack, we do not regard this type of attack, either.

However, to be able to targetedly manipulate a portion of the encrypted binary, in the first place, it must be decrypted using the correct k_u. Since k_u is recovered from π using the encrypted program image itself as part of the PUF-input, the binary must not be tampered with, before it is loaded for execution. Ignoring this will lead to the recovery of a wrong k_u, thereby invalidating the whole binary and consequently preventing its execution. Therefore the attack can only be accomplished, manipulating the encrypted binary in memory *after* it has been loaded for execution by the processor. This requires exact timing in combination with some control over or exact knowledge about the processors execution state and full control over the memory segment to be tampered with. Moreover, the knowledge of the position of certain clear-text opcodes inside the encrypted binary image can only be acquired by additional sophisticated attacks. The state of processor and memory has to be monitored during the execution of a binary. Since the clear-text binary is not known at this point to the attacker, he needs to progressively deduct his knowledge about the control flow of the binary. Methods to accomplish that may be monitoring well-known memory access pattern for relevant operations and analysing ALU-activity. The clear-text opcode to an operation identified to reside at a specific position in the binary is public information dependent on the architecture only. All those weaknesses that need to be exploited in combination to make the attack actually possible, may be countered by individual measures. Those include but are not limited to Oblivious RAM to obfuscate memory access patterns and payload data, and operation obfuscation by firing multiple random ALU operations at once, preventing detection methods for ALU-operations. The application of those measures to a secure execution environment have been described by Ascend [9, 36]. Those measures may be desirable anyway to protect payload data from being extracted even if data encryption is put in place. An additional conventional ISR-layer, obfuscating even the clear-text opcodes, may further hinder attacks.

Even if such an attacker was successful, at least the usage of the PUF prevents the predictability of the pad resulting from the CTR-mode decryption operation. This restricts the attackers immediate gain to be able to modify exactly this one position of a binary and only on-the-fly at execution time in a TOCTOU manner. He cannot deduct the key material used to generate the pad, independent from the number of pad-bits he knows and Therefore the attack needs to be repeated in its whole complexity for each and every memory location of the binary. We question the feasibility of this attack and like to point out in this context that the primary goal of the adversary model are injection attacks like those exploiting buffer overflows. Moreover, ROP attacks possible on SEPP are significantly less powerful than they are in general as mentioned earlier in this security evaluation. Offline attacks targeting the dormant image of the binary in memory are only secondary goals.

We have stated in the adversary model (Sec. 3) that we do not address hardware tampering directly and assume the chip itself as tamper-proof. Nevertheless SEPP promotes the goal of producing a tamper-proof device by minimizing the chip area that has to be trusted even in an adverse environment. Only there some intermediate values and keys like k_u and k_p need to be stored temporarily during execution. Those values are stored in protected internal registers which can only be read-out using sophisticated hardware attacks targeting this very small part of the CPU's internals. We excluded such an attack from the adversary model as we assume that typical attackers on embedded systems do not have the necessary skills and equipment. However, we argue that our approach is in favour of any additional measures to be taken to mitigate attacks which involve hardware tampering. Since SEPP utilizes a PUF, no permanently stored key values need to be protected in powered-off state of the processor, since there are no such values that may not be public. The extremely small chip area, we need to be trusted and confidential during runtime, essentially is the instruction fetch logic including the PUF and its wrapper.

6.2 Performance

Calculating the performance penalty: Decryption latency and bandwidth directly impact the processor's performance twofold: First, 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. Second, in a sequential stream of instructions, the processor has to be stalled when the bandwidth of the decryption module is smaller than the processor's bandwidth; we call this *execution latency.* The entire decryption process can be implemented very efficiently in hardware. As described in Sec. 5, our prototype induces no execution latency.

The only remaining time overhead during execution is the warm-up latency at the beginning of new basic blocks. The overall performance penalty, therefore, is dependent on the control flow of the program and 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.* These values are identical across SEPP and the baseline processor. The overhead can be calculated by $BIC \cdot lat_w$ as the product of the number of branching instructions and the warm-up latency in clock cycles.

SEPP's lat_w is 13 cycles due to the utilized AES-core implementation. Actually, this latency is reduced by the average common memory access latency, since the encryption pad is computed in parallel to the memory access. This decreases the effective lat_w to 8 cycles instead of 13 cycles per branch. For a hypothetical program with 1 mio. instructions, 10% branching instructions and a CPI of 1.5, the runtime penalty calculates to 1.53. The average memory access latency and CPI are estimations based on the OR1200's data sheet and system simulations.

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 Tab. 2 (Appendix).

A custom test program, we used, induces jump instructions by a single function call within a loop. This program consists of 14.3% branching instructions resulting in a runtime penalty of 1.84 compared to its execution on the baseline system. By replacing the function call by the function code only (including stack frame allocation and parameter passing) inside the loop, two out of three jumps are removed, leaving the loop jump. This results in 5.3% branching instructions and a reduced runtime penalty of 1.34, which 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. Therefore it is to be expected that unrolling a loop results in a significant speed-up of encrypted execution. To verify this assumption, we developed another short application containing a nested loop; an outer loop with a large number of iterations and an inner loop with a small number of iterations. We then compared the runtime of the unmodified program with a modified version. Therein the inner loop has been unrolled manually by removing the loop instructions and repeating the instructions in the loop body the appropriate number of times. This optimization reduces the percentage of branching instructions from 9.8 to 3.8 and thereby speeds up encrypted execution by factor 1.83, while unencrypted execution on the baseline system only gains a factor of approximately 1.33. The runtime penalty of the SEPP version of the program compared to the baseline version is reduced tremendously from 1.61 to 1.17 by the optimization.

The positive influence of unrolling loops is also evident in the following CoreMark[5] benchmarks, which we performed in order to enable comparison with future developments and other platforms, and to show the effects of our instruction-level decryption on actual calculations. We conducted the benchmark both, on our system in encrypted form, as well as on the baseline architecture in unencrypted form. It was compiled with four different GCC optimization settings: `-O0`, `-O2`, `-O3` and `-O3 -funroll-all-loops`. While `-O0` tells the compiler not to optimize at all, optimization level `-O2` activates all optimizations that do not increase code size, and `-O3` adds function inlining and register renaming. The flag `-funroll-all-loops` additionally enables loop unrolling. Fig. 7(a) compares the benchmark results of both systems for the different GCC settings. The warm-up latency is significant, but reasonable, resulting in a highest measured runtime penalty of approximately 49.4% when compiled with `-O2`. Compilation with `-O3` results in a 48.8% penalty and compilation with `-O3 -funroll-all-loops` in 43.5% penalty. When compiled with no optimization at all (`-O0`), the encrypted execution is 31% slower compared to the baseline processor, however, the baseline's performance is significantly reduced as well.

CoreMark results confirm that reduction of branching instructions result in a larger improvement on SEPP than on the baseline processor. Our prototype clearly profits more from loop-unrolling optimizations with `-funroll-all-loops` than the original system. Fig. 7(b) illustrates that our architecture benefits significantly more from loop unrolling – with a speed-up of 21% – than the baseline configuration, with only 9% speed-up.

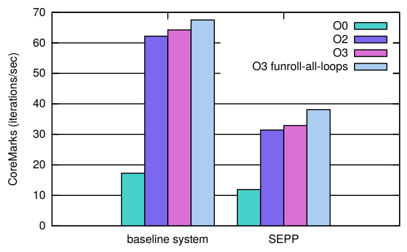
[5] `http://www.eembc.org/coremark` accessed on 09/02/2015

As performance solely depends on the warm-up latency of the deployed block cipher, recent advances in the design of hardware block ciphers promise a severe reduction of this latency. Specialized low-latency block ciphers like PRINCE [5] can perform encryption in a single cycle at a competitive area cost. In our mode of operation the block cipher implementation needs only to support encryption to produce pads which are used as a key stream for both en- and decryption. While code optimizations can boost the performance of the current design, we believe that the substitution of AES with a low-latency cipher would make the performance overhead almost negligible.

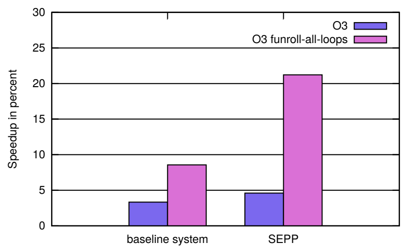
Unfortunately, the comparison of our prototype with related implementations is not straightforward, as authors in this field use a variety of methods for performance assessment. The AEGIS [28] developers used the SPEC2000 CPU[6] benchmark suite, we had not available. AEGIS can be operated in two different modes, namely Tamper-Evident (TE) processing or Private Tamper-Resistant (PTR) processing. With TE processing, the authors measure a performance degradation of up to 50% in the worst case, while other applications run with as little as 15% degradation. For PTR processing, their memory encryption causes up to 25% degradation, resulting in an overall performance penalty of up to 60%; while the authors state that in most cases the degradation stays below 40%. The authors additionally suggest that the performance can be increased by a larger L2 cache or L2 cache block size. As the overhead originates from the decryption of data and instructions during memory access, we argue that it can not be improved as easily as SEPP performance. However, it must be acknowledged that AEGIS processes both data and instructions, while SEPP only processes encrypted instructions.

The authors of the OASIS [26] instruction set extension only provide absolute time overheads for specific platform operations and compare their approach with other Trusted Platform Module benchmarks, which does not allow direct comparison to our benchmarks. The XOM [19] research paper provides a theoretical performance analysis, similar to our formal analysis. The authors state that

[6] `https://www.spec.org/cpu2000/`, accessed on 11/02/2015



(a) Results



(b) Speed-up by optimizations compared to O2

Fig. 7. CoreMark benchmark dependent on GCC optimization parameters

for one of their processor examples, the slow-down is less than 50% according to calculations. Similarly to the AEGIS approach, the XOM performance overhead originates from the memory pipeline, and is therefore presumably harder to improve than the SEPP performance.

We conclude that the runtime penalty of SEPP lies well within the 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.

7 Future Work

Implementing the processor using a true challenge-response PUF could provide insights about the utility of different kinds of PUFs for the current concept. Especially to compare the complexity of the management of challenges and responses and the corresponding helper data might support our approach.

Despite its drawbacks, we have chosen CTR-mode for the instruction encryption and decryption for its ease of implementation and straightforward understanding of its function. Thus we were able to rapidly prototype the proof-of-concept implementation to be able to show the general feasibility of our approach. With reasonable effort, CTR-mode can be exchanged for a number of other block cipher modes having the required properties. Those properties are known from disc encryption approaches, where an encryption block is not chained with the previous plain-text block whereby enabling random access. Candidates for the usage by SEPP are the block-cipher modes devised by *Liskov, Rivest, and Wagner* (LRW) [20], being an improved security-complexity trade-off, or the *XEX-based tweaked-codebook mode with ciphertext stealing* (XTS) [1].

If we want to use our approach in distributed scenarios like cloud computing, but also for updating software in already deployed embedded systems, we need to provide a secure channel for the transmission of k_u. This can be achieved by embedding an asymmetric cryptography module into the secure part of SEPP.

To enable software updates in the field, a binary $\mathcal{B}$ initially deployed in a secure environment as stated before, may contain an update function written as part of the secure binary $\mathrm{enc}_{k_u}(\mathcal{B})$. This update function needs to be able to take a user-key ${k_u}'$ and an encrypted binary $\mathrm{enc}_{{k_u}'}(\mathcal{B}')$ and feed it into the PUF-driven executable-image generation process of SEPP. SEPP will return $\pi' = \mathrm{enc}_{{k_p}'}({k_u}')$ to be packaged into a new image $\mathrm{img}(\mathcal{B}', \pi')$ for the current SEPP instance. A user can send an encrypted binary $\mathrm{enc}_{{k_u}'}(\mathcal{B}')$ and ${k_u}'$ to this function, possibly from remote via an untrusted network connection. ${k_u}'$ needs to be transmitted securely. The update function already deployed in binary $\mathcal{B}$ must therefore contain a suitable encryption scheme, e. g. RSA, for a secured channel between user and secure binary. Since k_u can not be retrieved outside of the IF phase of the processor, k_u does not need to remain the same for both, the already deployed and the about-to-be deployed application. Normally the PUF-driven executable-image generation process of SEPP, generating an encrypted $\pi = \mathrm{enc}_{k_p}(k_u)$, should be deactivated after deployment of the binaries. 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 a trade-off for a convenient update path. Nevertheless, the image generation process can be restricted to be only usable when called from within a program running in encrypted mode. This is due to the update function being part of a legitimately encrypted binary already running on a SEPP instance. To ensure the injection prevention security goal, this restriction of the image generation must be enforced. There remains the remote possibility that this might be exploitable by an attacker to encrypt his own code for the current processor instance. This, however, should be quite difficult to actually perform, since a set of code to be validly decrypted in the security context of the legitimate application has to be injected at some point in that application. This as for itself is prevented by the security properties of SEPP in the first place, as laid out before.

Besides these enhancements, our concept may be augmented in future work to provide for a larger variety of security demands. This includes data encryption which might be accomplished via homomorphic encryption [32], Oblivious RAM [27] or Authenticated Storage [33].

A further open question about data confidentiality is the decryption of constant data sections in the binary. A dedicated hardware extension of SEPP could be devised in future work that will reduce any performance penalties of those decryption processes. Using an approach with minimal architectural changes, the compiler would encrypt any constant values in the binary and automatically add a suitable decryption function to the binary. This, however, will come at the cost of increased compiler customization.

Beyond, we also need to consider additional requirements from other scenarios, e. g. the need to interoperate with virtualization mechanisms in cloud computing scenarios.

8 Conclusion

In this paper, we have presented SEPP, an architecture that embeds a PUF-based decryption module deeply into a CPU design in order to prevent injection of malicious code or reverse engineering of programs in embedded systems. Code gets encrypted and thus bound to single individual CPU instances. This may also open up additional opportunities, e. g. for offering additional features to customers but preventing those features from being activated in other devices.

As our evaluation has shown, we reached the envisioned security goals while at the same time incurring only a reasonable performance penalty on our prototypical implementation on an FPGA. Remaining attack vectors require either exceedingly sophisticated hardware attacks, or are based on highly unlikely ROP scenarios. Compared to previously proposed solutions, this constitutes a significant step forward in the level of security and it paves the way for improved performance.

Our future work will focus on the completion of the development environment, operating system support for our platform by porting a Linux system to it, further performance enhancements, and addressing challenges in additional scenarios, e. g. virtualization in cloud computing.

Acknowledgment

We want to thank Henning Kopp, Rens van der Heijden and Christoph Bösch for their valuable input on our approach.

References

[1] Shakil Ahmed, Khairulmizam Samsudin, Abdul Rahman Ramli, and Fakhrul Zaman Rokhani. "Effective implementation of AES-XTS on FPGA". In: *TENCON 2011 - 2011 IEEE Region 10 Conference*. Nov. 2011, pp. 184–186.

[2] T. Alves and D. Felton. *Trustzone: Integrated hardware and software security.* white paper. ARM, July 2004.

[3] Frederik Armknecht, Roel Maes, Ahmad-Reza Sadeghi, Francois-Xavier Standaert, and Christian Wachsmann. "A formal foundation for the security features of physical functions". In: *SP*. IEEE, 2011.

[4] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, and Darko Stefanović. "Randomized Instruction Set Emulation". In: *ACM TISSEC* 8.1 (Feb. 2005), pp. 3–40.

[5] Julia Borghoff et al. "PRINCE – A Low-Latency Block Cipher for Pervasive Computing Applications". English. In: *Advances in Cryptology – ASIACRYPT 2012*. Ed. by Xiaoyun Wang and Kazue Sako. Vol. 7658. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 208–225.

[6] Christoph Bösch, Jorge Guajardo, Ahmad-Reza Sadeghi, Jamshid Shokrollahi, and Pim Tuyls. "Efficient helper data key extractor on FPGAs". In: *CHES*. IACR, 2008.

[7] Martin Bossert. *Channel Coding for Telecommunications.* New York: John Wiley & Sons, 1999.

[8] Crispin Cowan et al. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks". In: *Proceedings of the 7th Conference on USENIX Security Symposium.* Vol. 7. SSYM'98. San Antonio, Texas: USENIX Association, Jan. 1998, pp. 5–5.

[9] Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. "A Secure Processor Architecture for Encrypted Computation on Untrusted Programs". In: *STC*. ACM, 2012.

[10] Jorge Guajardo, Sandeep S Kumar, Geert Jan Schrijen, and Pim Tuyls. "FPGA Intrinsic PUFs and Their Use for IP Protection". In: *CHES*. IACR, 2007.

[11] Matthias Hiller, Dominik Merli, Frederic Stumpf, and Georg Sigl. "Complementary IBS: Application Specific Error Correction for PUFs". In: *HOST*. IEEE, 2012.

[12] Matthias Hiller and Georg Sigl. "Increasing the Efficiency of Syndrome Coding for PUFs with Helper Data Compression". In: *DATE*. ACM/IEEE, 2014.

[13] Matthias Hiller, Michael Weiner, Leandro Rodrigues Lima, Maximilian Birkner, and Georg Sigl. “Breaking through Fixed PUF Block Limitations with Differential Sequence Coding and Convolutional Codes”. In: *TrustED*. ACM, 2013.

[14] IBM. *GCC extension for protecting applications from stack-smashing attacks*. Aug. 2005. URL: `http://www.research.ibm.com/trl/projects/security/ssp` (visited on 08/25/2014).

[15] Stefan Katzenbeisser et al. “PUFs: myth, fact or busted? a security evaluation of physically unclonable functions (PUFs) cast in silicon”. In: *CHES*. IACR, 2012.

[16] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. “Countering Code-injection Attacks with Instruction-Set Randomization”. In: *Proceedings of the 10th ACM Conference on Computer and Communications Security*. CCS ’03. Washington D.C., USA: ACM, Oct. 2003, pp. 272–280.

[17] Ralph Langner. *To Kill a Centrifuge - A Technical Analysis of What Stuxnet’s Creators Tried to Achieve*. Tech. rep. Nov. 2013.

[18] Vincent van der Leest, Bart Preneel, and Erik van der Sluis. “Soft Decision Error Correction for Compact Memory-Based PUFs Using a Single Enrollment”. In: *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Ed. by Emmanuel Prouff and Patrick Schaumont. Vol. 7428. LNCS. Springer Berlin / Heidelberg, 2012, pp. 268–282.

[19] David Lie et al. “Architectural Support for Copy and Tamper Resistant Software”. In: *SIGOPS Operating Systems Review* 34.5 (Nov. 2000), pp. 168–177.

[20] Moses Liskov, Ronald L. Rivest, and David Wagner. “Tweakable Block Ciphers”. en. In: *Journal of Cryptology* 24.3 (Sept. 2010), pp. 588–613.

[21] Roel Maes, Pim Tuyls, and Ingrid Verbauwhede. “Low-overhead implementation of a soft decision helper data algorithm for SRAM PUFs”. In: *CHES*. 2009.

[22] Roel Maes, Anthony Van Herrewege, and Ingrid Verbauwhede. “PUFKY: A Fully Functional PUF-Based Cryptographic Key Generator”. In: *CHES*. 2012.

[23] Abhranil Maiti and Patrick Schaumont. “Improved Ring Oscillator PUF: An FPGA-friendly Secure Primitive”. In: *Journal of Cryptology* 24.2 (2011), pp. 375–397.

[24] Microsoft. *Next-Generation Secure Computing Base*. 2005. URL: `http://www.microsoft.com/resources/ngscb/default.mspx` (visited on 08/25/2014).

[25] Aleph One. “Smashing the Stack for Fun and Profit”. In: *Phrack* 7.49 (Nov. 1996).

[26] Emmanuel Owusu et al. “OASIS: On Achieving a Sanctuary for Integrity and Secrecy on Untrusted Platforms”. In: *CCS*. ACM, Nov. 2013.

[27] Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten Van Dijk, and Srinivas Devadas. “Design space exploration and optimization of path oblivious ram in secure processors”. In: *ISCA*. ACM, 2013.

[28] Edward G Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. “AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing”. In: *ICS*. ACM, June 2003.

[29] Edward G Suh, Charles W O’Donnell, Ishan Sachdev, and Srinivas Devadas. “Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions”. In: *SIGARCH Computer Architecture News* 33.2 (May 2005), pp. 25–36.

[30] Gookwon Edward Suh and Srinivas Devadas. “Physical Unclonable Functions for Device Authentication and Secret Key Generation”. In: *ACM/IEEE Design Automation Conference (DAC)*. 2007, pp. 9–14.

[31] Trusted Computing Group. *TCG Specification Architecture Overview Revision 1.2*. 2004. URL: `http://www.trustedcomputinggroup.com/home` (visited on 08/25/2014).

[32] N.G. Tsoutsos and M. Maniatakos. “HEROIC: Homomorphically EncRypted One Instruction Computer”. In: *DATE*. Mar. 2014.

[33] Hsin-Jung Yang, Victor Costan, Nickolai Zeldovich, and Srinivas Devadas. “Authenticated Storage Using Small Trusted Hardware”. In: *CCSW*. ACM, 2013.

[34] Jun Yang, Youtao Zhang, and Lan Gao. “Fast Secure Processor for Inhibiting Software Piracy and Tampering”. In: *MICRO*. IEEE/ACM, Dec. 2003.

[35] Meng-Day Yu and Srinivas Devadas. “Secure and Robust Error Correction for Physical Unclonable Functions”. In: *IEEE Design & Test of Computers* 27.1 (2010), pp. 48–65.

[36] Xiangyao Yu, Christopher W. Fletcher, Ling Ren, Marten van Dijk, and Srinivas Devadas. “Generalized External Interaction with Tamper-resistant Hardware with Bounded Information Leakage”. In: *CCSW*. ACM, 2013.

Appendix

		$\frac{2}{3}$ 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

Table 2. Results of custom tests executed on prototype implementation