# Design of the Secure Execution PUF-based Processor (SEPP)

Stephan Kleber[1], Florian Unterstein[1], Matthias Matousek[1], Frank Kargl[1], Frank Slomka[3], and Matthias Hiller[4]

[1] Institute of Distributed Systems, Ulm University, Germany, stephan.kleber@uni-ulm.de, florian.unterstein@aisec.fraunhofer.de, matthias.matousek@uni-ulm.de, frank.kargl@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 is its vulnerability to code injection attacks. Equally unsolved is the susceptibility of software to reverse engineering, which undermines code confidentiality. We propose an approach that solves both kinds of security problems by employing instruction-level code encryption combined with the use of a physical unclonable function (PUF). Our *Secure Execution PUF-based Processor* (SEPP) architecture is designed to minimize the attack surface, as well as the performance impact, and requires no significant changes to the software development process. Our approach supports distributed systems, as the secure execution environment needs not be physically available to the developer.

## 1 Introduction

Code injection, especially when performed remotely, is one of the most effective strategies for malicious attackers. Since the infamous phrack article "Smashing The Stack For Fun And Profit" [8] 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. It is still an open security challenge to effectively prevent injection of unauthorized code into an execution environment.

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 the *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. Not properly encrypted, code will not execute successfully. Therefore it cannot be injected remotely.

## 2 Related Work

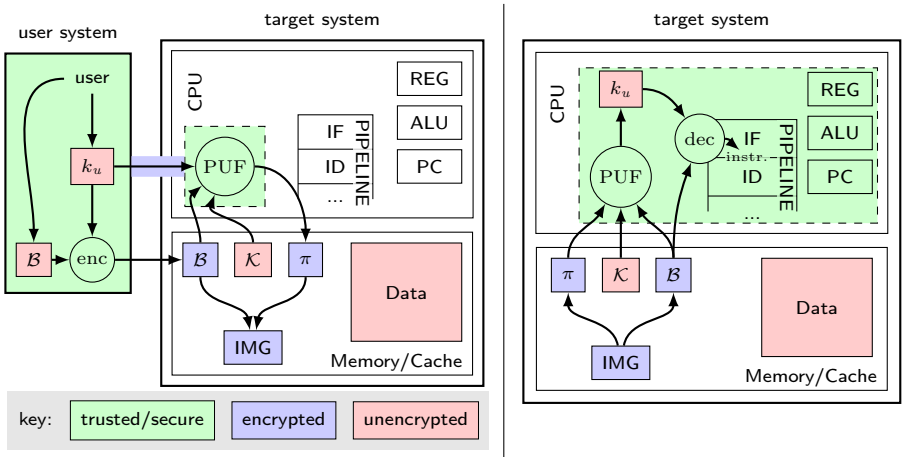
To achieve code injection protection and code confidentiality, it is necessary to maintain control over the execution environment, even if it is physically accessible to an adversary. Previous approaches used the term isolated execution environment (IEE) [9]. We relate our work to the closest IEE concepts: Compared to the *execute-only memory* (XOM) architecture [6], our approach does not require the assumption that main memory or even caches are secure. The *AEGIS* secure processor [10, 11] is the first attempt to utilize the challenge-response (CR) behavior of a PUF. AEGIS requires extensive compiler and OS support, as well as additional modified hardware like a custom memory controller. Our approach aims for a smaller trusted computing base (TCB) and better compatibility with existing code. *OASIS* [9] establishes data confidentiality and integrity with cryptographic keys bootstrapped from a PUF. However, OASIS does not encrypt the code itself. In general, we aim for a deeper embedding where code remains encrypted in memory and caches and gets decrypted just within the execution pipeline. Moreover, our design aims at minimizing not only the TCB, but also the necessity for changes to known programming models.

*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 CR protocol [1, 5]. Instead of storing secrets permanently, PUFs reveal their secret only during runtime. Popular PUF types for key generation are the SRAM PUF [3] and the Ring-Oscillator (RO) PUF [7]. In this work, we use a Complementary Index-Based Syndrome coding (C-IBS) RO PUF implementation [4].

## 3 Adversary Model

There are two distinct addressed adversary models: For the **code confidentiality** scenario (e.g., to protect intellectual property in embedded systems), we assume that the attacker has physical access to the processor and its peripheral connections like memory bus lines. The attacker tries to learn parts of the application code by reading out memory or registers. For the **malicious code injection** scenario, we assume an attacker has the ability to place arbitrary data into the processors main memory.

Attacks like denial-of-service (DoS), e. g. injecting random invalid instructions, and hardware side-channels are considered subjects of complementary research such as Ascend [2]. 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, like 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.




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

Fig. 1: SEPP architecture

## 4 Secure Execution PUF-based Processor Architecture

The SEPP architecture consists of two major components: The execution environment, where code is securely executed, which we call *target system*. Its core is a processor providing runtime decryption of single instructions directly within the execution pipeline. It includes strict hardware binding of code through utilization of a PUF. For this, we extended the OpenRISC OR1000 architecture by a PUF module and an instruction decryption module. The other component is the development machine which we call *user system*. On the user system, the user generates programs that will be deployed on the target system.

After a user has compiled code into a program binary on the trusted user system, he needs to encrypt the binary. The program code is encrypted on basic block level, using a symmetric cipher in CTR mode, enabling random access to support execution branches. For this symmetric encryption, a key $k_u$ is chosen by the user. Each basic block is encrypted using $k_u$ with the CTR mode nonce set to the virtual address of the beginning of the basic block. Starting from zero, the counter is incremented while the nonce stays the same for a basic block.

How a program binary $\mathcal{B}$ is bound to a hardware instance is shown in Fig. 1a. The encrypted binary $\text{enc}_{k_u}(\mathcal{B})$ and $k_u$ are thereto transmitted to the target system. In this, $k_u$ must remain confidential. By hashing, the encrypted binary and the security kernel $\mathcal{K}$ are bound together. This hash constitutes the challenge $c = \text{Hash}(\mathcal{K}, \text{enc}_{k_u}(\mathcal{B}))$. $c$ is then used as input to the PUF, using its properties to bind $\mathcal{K}$ and $\mathcal{B}$ to the processor by generating the cryptographic key $k_p = \text{PUF}(c)$ as a response. To protect the user key $k_u$, required for program-execution, it is encrypted using this $k_p$. This public representation of $k_u$ we call $\pi = \text{enc}_{k_p}(k_u)$. This has the advantage that the target hardware is not needed in order to prepare a binary while retaining the desired security properties.

To minimize the parts of the processor that are required to be trusted, the decryption module is included directly in the instruction fetch stage of the processor's pipeline. The module continuously pre-computes a keystream during the execution of a basic block. This keystream is XORed with the incoming encrypted instructions, decrypting them. A decrypted instruction is then forwarded to the instruction decode stage as usual.

To remain self-contained, the correct $k_u$ must to be recovered for execution only from the encrypted $\pi$ inside the program image and $k_p$. The target system needs not store any program-specific values. Only the same processor instance is able to generate the correct $k_p$ to restore $k_u = \mathrm{dec}_{k_p}(\pi)$. The RO PUF implementation we utilized for our approach does not provide CR behavior but generates a single fixed response. CR behavior, however, is required for the generation of cryptographic keys which are not only bound to the device but also to the program binary. We constructed a CR wrapper around the device specific RO PUF-output $s_p$ as an alternative to a CR PUF, so $k_p = \mathrm{PUF}(c) = \mathrm{Hash}(s_p, c)$.

## 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 the OpenRISC Reference Platform (ORPSoC) which is an implementation of OR1000, an open-source RISC architecture with a 32 bit wide instruction set and a five stage, single issue pipeline. 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 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 kernel, i. e., the generation and execution of encrypted program images in interaction with the PUF module.

## 6 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 and reverse engineering of programs in embedded systems. Code is encrypted and thereby bound to single individual CPU instances. As our prototype shows, the envisioned security goals of SEPP can be reached. 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, evaluations and performance enhancements.

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

## References


[1] Frederik Armknecht, Roel Maes, Ahmad-Reza Sadeghi, Francois-Xavier Standaert, and Christian Wachsmann. "A Formal Foundation for the Security Features of Physical Functions". In: *Security and Privacy (S&P). Symposium on*. IEEE, 2011.

[2] Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. "A Secure Processor Architecture for Encrypted Computation on Untrusted Programs". In: *Scalable Trusted Computing. Proceedings of the Seventh Workshop on*. ACM, 2012.

[3] Jorge Guajardo, Sandeep S. Kumar, Geert Jan Schrijen, and Pim Tuyls. "FPGA Intrinsic PUFs and Their Use for IP Protection". In: *Cryptographic Hardware and Embedded Systems (CHES). 9th International Workshop on*. IACR, 2007.

[4] Matthias Hiller, Dominik Merli, Frederic Stumpf, and Georg Sigl. "Complementary IBS: Application Specific Error Correction for PUFs". In: *Hardware-Oriented Security and Trust (HOST). International Symposium on*. IEEE, 2012.

[5] Stefan Katzenbeisser, Ünal Kocabaş, Vladimir Rožić, Ahmad-Reza Sadeghi, Ingrid Verbauwhede, and Christian Wachsmann. "PUFs: Myth, Fact or Busted? A Security Evaluation of Physically Unclonable Functions (PUFs) Cast in Silicon". In: *Cryptographic Hardware and Embedded Systems (CHES). 14th International Workshop on*. IACR, 2012.

[6] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. "Architectural Support for Copy and Tamper Resistant Software". In: *SIGOPS Operating Systems Review* 34.5 (Nov. 2000), pp. 168–177.

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

[8] Aleph One. "Smashing the Stack for Fun and Profit". In: *Phrack* 7.49 (Nov. 1996). URL: `http://www.phrack.com/issues.html?issue=49&id=14` (visited on 08/24/2015).

[9] Emmanuel Owusu, Jorge Guajardo, Jonathan McCune, Jim Newsome, Adrian Perrig, and Amit Vasudevan. "OASIS: On Achieving a Sanctuary for Integrity and Secrecy on Untrusted Platforms". In: *Conference on Computer and Communications Security (CCS)*. ACM, Nov. 2013.

[10] Edward G. Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing". In: *International Conference on Supercomputing. Proceedings of the 17th annual*. ACM, June 2003.

[11] 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.