Secure Data-Binding in FPGA-based Hardware Architectures utilizing PUFs

Florian Frank
Florian.Frank@uni-passau.de
University of Passau
Passau, Germany

Martin Schmid
Martin.Schmid@uni-passau.de
University of Passau
Passau, Germany

Felix Klement
Felix.Klement@uni-passau.de
University of Passau
Passau, Germany

Purushothaman Palani
purushothsankari@vt.edu
Virginia Tech
Blacksburg, USA

Andreas Weber
Andreas.Weber@uni-passau.de
University of Passau
Passau, Germany

Elif Bilge Kavun
Elif.Kavun@uni-passau.de
University of Passau
Passau, Germany

Wenjie Xiong
wenjiex@vt.edu
Virginia Tech
Blacksburg, USA

Tolga Arul
Tolga.Arul@uni-passau.de
University of Passau
Passau, Germany

Stefan Katzenbeisser
Stefan.Katzenbeisser@uni-passau.de
University of Passau
Passau, Germany

ABSTRACT

In this work, a novel FPGA-based data-binding architecture incorporating PUFs and a user-specific encryption key to protect the confidentiality of data on external non-volatile memories is presented. By utilizing an intrinsic PUF derived from the same memory, the confidential data is additionally bound to the device. This feature proves valuable in cases where software is restricted to be executed exclusively on specific hardware or privacy-critical data is not allowed to be decrypted elsewhere. To improve the resistance against hardware attacks, a novel method to randomly select memory cells utilized for PUF measurements is presented. The FPGA-based design presented in this work allows for low latency as well as small area utilization, offers high adaptability to diverse hardware and software platforms, and is accessible from bare-metal programs to full Linux kernels. Moreover, a detailed performance and security evaluation is conducted on five boards. A single read or write operation can be executed in 0.58 μs when utilizing the lightweight PRINCE cipher on an AMD Zync 7000 MPSoC. Furthermore, the entire architecture occupies only about 10% of the FPGA's available space on a resource-constrained AMD PYNQ-Z2. Ultimately, the implementation is demonstrated by storing confidential user data on new generations of network base stations equipped with FPGAs.

CCS CONCEPTS

• **Computer systems organization → Reconfigurable computing**; • **Hardware → Reconfigurable logic applications**; • **Security and privacy → Hardware-based security protocols**.

ASIA CCS '24, July 1–5, 2024, Singapore, Singapore

ACM ISBN 979-8-4007-0482-6/24/07
https://doi.org/10.1145/3634737.3656996

KEYWORDS

Hardware Security, FPGAs, Physical Unclonable Functions, Privacy, Broadband Cellular Networks

ACM Reference Format:
Florian Frank, Martin Schmid, Felix Klement, Purushothaman Palani, Andreas Weber, Elif Bilge Kavun, Wenjie Xiong, Tolga Arul, and Stefan Katzenbeisser. 2024. Secure Data-Binding in FPGA-based Hardware Architectures utilizing PUFs. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '24), July 1–5, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3634737.3656996

1 INTRODUCTION

During the last years, Field-Programmable Gate Arrays (FPGAs) turned from niche products mostly used in research and development settings to increasing adoption as integral components across various domains, including data centers [12, 28], network devices [18, 26], or automotive electronics [17]. This development is caused by improvements in performance, energy consumption, and the number of available logical cells in newer generations of FPGAs. Their primary advantage is the ability to program custom hardware designs, which can be reconfigured even during run-time, resulting in a high degree of flexibility, remarkable performance improvements, and significant cost savings compared to the development of Application-Specific Integrated Circuits (ASICs). Despite these benefits, FPGAs deployed in the field, e.g., in a cellular network base station, are vulnerable to physical attacks. For example, the attacker can probe the memory or peripherals to gain privacy-sensitive data processed on the FPGA. Furthermore, an attacker could also modify or replace certain components of an FPGA platform. To counteract such attacks, additional measures beyond a simple decryption of data on the device are necessary.

One approach to mitigate these vulnerabilities is to bind data encryption to specific hardware and restrict processing to the device. Maintaining the security of that single device and, therefore, the critical data processed on it becomes a much more manageable

task than in highly heterogeneous and distributed systems. In general, hardware-data binding provides several benefits. It improves overall security and ensures that data can only be processed on authorized and trusted devices. Moreover, tamper resistance can be enhanced by ensuring that the dedicated device offers security features like hardware-based encryption or a secure boot process. Furthermore, it simplifies the enforcement of access restrictions and controls accountability when accessing sensitive data by ensuring that the data processing is exclusively permitted for authorized applications and users. Data-binding has been demonstrated in various use cases, such as maintaining the integrity of Digital Rights Management (DRM)-systems [36], the prevention of unauthorized replication of software, or the secure storage of highly critical biometric, healthcare, military, or financial data [30]. This method prevents the data from disclosure or tampering attacks by allowing only the decryption on a single physical device.

This work presents an FPGA-based data-binding architecture utilizing intrinsic memory-based Physical Unclonable Functions (PUFs) to tackle the above-mentioned problems while, at the same time, offering notable latency reductions, improved flexibility and security against hardware attacks exploiting the advantages of optimized hardware designs.

PUFs exploit unique variations occurring during the manufacturing process of hardware components such as memory modules to generate unpredictable digital fingerprints. PUFs accept a challenge denoted as c and generate a response r, based on c. Notably, the same challenge sent to a device leads to the same response on a specific device, whereas sending the same challenge to a different device leads to a completely different response. These properties allow the use of these responses for secure device authentication and identification or, as presented in this paper, the generation of cryptographic keys to bind data to a designated device. Particularly intrinsic PUFs, for example, such based on memory modules, offer a cost-effective solution for key generation in resource-constrained devices by exploiting marginal variations inherent in existing hardware. This approach eliminates the need for memory-intensive key management, which could additionally be prone to side-channel attacks. Moreover, no additional hardware such as Trusted Platform Modules (TPMs) is required while offering a substantial set of keys defined by the specific set of challenges and corresponding responses. Furthermore, some PUF implementations provide tamper detection capabilities [16].

Our design makes use of these benefits and leverages a PUF retrieved from the same memory that stores the privacy-sensitive data and uses the PUF-response combined with an externally provided cryptographic key to encrypt the data on the memory. This method ensures that data can only be decrypted by measuring the PUF of the memory that hosts the data and PUF. Prior to conducting a PUF measurement, the content of the selected cells is temporarily stored in volatile registers on the FPGA, allowing the acquisition of PUF measurements on cells already containing other data. Unlike other commonly used memory-based PUF constructions, as cited in the references [39, 47], our solution eliminates the need for a dedicated reserved area. This enables the utilization of the entire memory capacity to store payload data, while allowing the execution of PUF measurements across all regions of the memory module.

Furthermore, even if an attacker gains physical access to the device and manages to read out the PUF, he cannot decrypt the information without possessing an additional external key. To increase the resistance against hardware attacks, we additionally present a novel method, which obfuscates the locations of the used PUF cells by spreading them across the logical memory space based on the externally provided cryptographic key, also used when encrypting the memory. Unlike a hard-coded manufacturer-provided key, our random PUF cell selection enables key revocation by choosing a different set of random cells. Moreover, this measure prevents a PUF-readout even if an attacker possesses the device.

Our FPGA-based data-binding architecture is implemented on devices incorporating Multi-Processor System-on-Chips (MPSoCs) consisting of one or multiple CPUs and an FPGA to accelerate specific tasks. In contrast to prior works and well-established disk encryption schemes, our work provides several advantages:

We do not require any additional or dedicated hardware to achieve secure hardware data binding. Moreover, a high degree of flexibility is offered through our modular design approach. In addition to the modular design, all buses and address buses can be customized to match the capabilities of high-level resource-constrained devices. Overall, our design is optimized for high performance and low area consumption, achieved through the hardware implementation of optimized lightweight ciphers. To achieve all these benefits, our design is encapsulated within the FPGA accessible from the CPU, from which different instructions to the FPGA such as storing the external secret, reading out the PUF, or storing and loading confidential data using the PUF-key and the external secret can be sent. Finally, we demonstrate our implementation in a practical use case by leveraging FPGAs in base stations of emerging generations of cellular networks.

1.1 Contributions

In this paper, we present the following novel contributions:

- Design of a data-binding architecture using intrinsic memory-based PUFs to provide confidentiality and data-binding of privacy-sensitive data on MPSoCs.
- The highly flexible hardware design allows the adoption of different use cases and implementation across a spectrum of devices, spanning from resource-constrained MPSoCs to high-end architectures. It also provides compatibility with different memory modules and memory-based PUF implementations.
- Implementation of a novel method to randomly distribute memory cells based on an external key used for PUF invocation, eliminating both the need for an internal entropy source and the impacts of a stolen device.
- Extensive performance and security evaluation on five devices with different performance capabilities to assess the presented implementation. The assessment includes an analysis of six variants of lightweight ciphers and a proof-of-concept implementation of the entire architecture supporting read- and write-latency PUFs.
- Demonstration of the implementation in a practical use case in the domain of new generations of cellular network base stations.

1.2 Paper Organization

Section 2 presents works related to data-binding and FPGA-based PUFs. In Section 3, the capabilities and limitations of an attacker are outlined, based on the previously presented design. Furthermore, in Section 4, the design and architecture of our data-binding architecture are presented. Section 5 presents important aspects of the implementation. In Section 6, an evaluation of the area occupied on the FPGA and the latency of the implementation is given. In Section 7, the security aspects of the implemented design are discussed. In Section 8, the implementation is demonstrated through a use case in the domain of new generations of mobile cellular networks. Finally, the main findings are summarized, and future research directions are presented in Section 9.

2 RELATED WORK

This section provides a comprehensive review of existing work related to the topic presented in this paper, including several FPGA architectures incorporating PUFs for hardware-software binding, Intellectual Propertys (IPs) protection, or FPGA-based enclaves.

Xilinx provides a PUF implementation as part of the Zync® UltraScale+™ platform [31]. This solution allows for encrypting user data on an external memory using the Advanced Encryption Standard (AES). Therefore, the PUF cannot be directly accessed. Instead, it requires the AES-based encryption process exclusively available on Zync® UltraScale+™ MPSoCs. In contrast, our implementation is highly adaptable to different use cases and hardware platforms thanks to exchangeable modules.

Kleber *et al.* [27] presents an implementation using PUFs to encrypt basic blocks of programs by one-time keys derived from FPGAs to bind program executables to specific hardware. The prototype implementation presented in this paper utilizes a PUFs based on Ring-Oscillators (ROs), but discusses the use of other PUF-types, for example, Static Random-Access Memory (SRAM) PUFs or Bistable-Ring PUFs. In general, this work is based on a rather heavy AES implementation instead of using lightweight ciphers. Further, the implementation is specifically tailored for basic block encryption, whereas we provide a generic solution suitable for a variety of use cases.

Sepulveda *et al.* [41] presents a PUF-based architecture for memory authentication and integrity verification on MPSoCs using an implementation called k-SUM PUF described in the work of Maiti *et al.* [29]. Therefore, the paper only focuses on the properties of integrity and verification and does not address data confidentiality, which is a key aspect of our work.

Zhao *et al.* [53] presents an Trusted Execution Environment (TEE) on FPGAs in cloud environments. Here, the TEE is used as Root of Trust for secure boot or in remote attestation. Further research on FPGAs-based enclaves is presented by the following works [20, 46]. Implementations protecting bit-streams of FPGAs using PUFs are outlined in the following papers [22, 48–50].

In addition, several studies have explored FPGA-based encryption and authentication protocols utilizing RO-PUFs [8, 9, 54]. A limitation of the majority of the above-described works stems from the adoption of this PUF type. RO-PUFs require two concurrent ROs to generate a single response bit. Therefore, the implementation of ROs to produce an adequate number of bits to serve as cryptographic keys occupies a significant amount of area on the FPGA. Moreover, the ROs must be placed as symmetric paths, which requires a manual placement on the FPGA, making these implementations inflexible. In contrast, our design enables the implementation of various types of intrinsic memory-based PUFs, significantly reducing the footprint on the FPGA. Moreover, our design binds the data to the memory rather than binding the data to the FPGA, allowing for flexible hardware upgrades and thus enhances the maintainability.

3 ATTACKER MODEL

In this section, we define the abilities and limitations of an adversary $\mathcal{A}$, from whom the encrypted data stored within the memory module should be kept confidential. This definition is used for conducting the security evaluation of our architecture in Section 7. We assume that $\mathcal{A}$ has knowledge of our architecture. The type as well as the configuration of the PUF that is used for encrypting the data in the system, except for the specific set of memory cells queried by the PUF, are known to $\mathcal{A}$. We define $\mathcal{A}$ to have the following abilities:

- Retrieval of the complete ciphertext stored in memory;
- Encryption of arbitrary data and storing the resulting ciphertext in memory at an arbitrary location;
- Modification and subsequent decryption of elements stored in memory.
- Evaluation of the PUF on arbitrary parameters;
- Theft of the memory module.

In addition to outlining the abilities of $\mathcal{A}$, our attacker model sets specific boundaries. Firstly, $\mathcal{A}$ does not know external secrets passed to the device, specifically the initialization secret that is defined in Section 4. Moreover, our attacker model excludes any possible attacks probing the buses within the MPSoC or between the MPSoC and the connected memory module. Consequently, the attacker could recreate the PUF response, having in-depth knowledge about the error correction mechanism and helper data. As a result, the security of the system would rely on the confidentiality of the initialization secret.

Under these assumptions, it must not be possible for $\mathcal{A}$ to break confidentiality by gaining any information on the data stored on the device or on the keys and exploiting tweaks used during the encryption process.

4 DESIGN AND ARCHITECTURE

The design of our architecture facilitates the extraction of PUFs from an off-chip non-volatile memory module also designated to store confidential data. Subsequently, for run-time encryption, two keys are used, one based on the PUF, as well as an externally provided cryptographic key, only known to the user. The encryption with these keys ensures both hardware data-binding and data confidentiality.

Furthermore, our goal is to optimize resource efficiency regarding area and time consumption while employing a modular approach adaptable and scalable to various use cases and hardware configurations. Therefore, our design and implementation target hybrid devices such as MPSoCs, comprising a CPU, which we refer to as the Processing System (PS), and an FPGA, which we refer to

as Programming Logic (PL). The majority of the components are implemented on the PL, allowing for an optimized hardware design and, thus, an execution with low latency and high flexibility. Moreover, the utilized memory module is accessed directly by the PL, supporting various PUF implementations on several memory modules using a customized hardware-implemented memory controller. Figure 1 provides an overview of the proposed architecture.

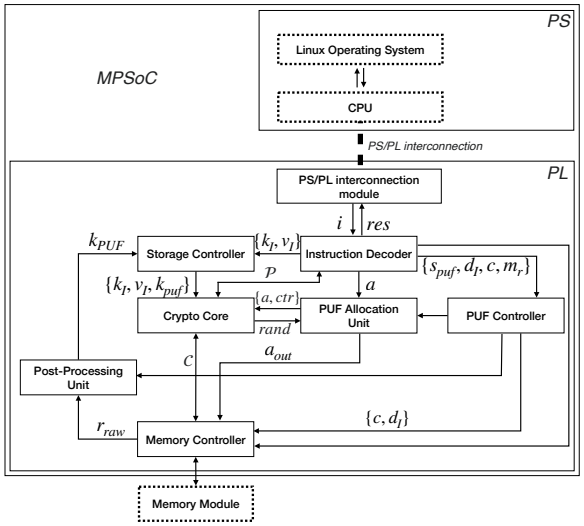

Figure 1: General architecture overview of the data-binding architecture.

As shown in Figure 1, our design achieves modularity by dividing it into several exchangeable and easy-to-maintain modules *m*, presented as rectangles with solid frames interacting through different connections *cn*, drawn as arrows. During startup, the user-possessed cryptographic key k_I and a random v_I must be transferred from the PS to the PL and stored by the *Storage Controller*. Afterward, a PUF measurement on a memory connected to the PL is performed by the *PUF Controller* returning a raw response r_{raw}, which is corrected by the *Post-Processing Unit*. Subsequently, the *Storage Controller* stores the corrected response k_{puf}. Therefore, different PUF implementations, identified by an identifier s_{puf}, can be implemented and selected at execution time. During this process, random memory cells are selected by the *PUF Allocation Unit* using the *Crypto Core* module, returning random values. The memory access is performed by a custom *Memory Controller*. After finishing the initialization phase with the PUF readout, the user application can send read- and write-requests from the PS to the PL to store confidential data, encrypted by two keys. These keys k_{puf} and k_I are forwarded to the *Crypto Core*, wrapping the cryptographic cipher. The *PS/PL Interconnection Module* is responsible for receiving instructions from the CPU and replying with corresponding answers. It forwards all instructions to the *Instruction Decoder*, responsible for decoding the commands, setting the required signals, and maintaining the synchronization between the modules. Before integrating the architecture into an application, the PUF must be enrolled, and PUF-dependent helper data *hd* is generated, which is stored in non-volatile memory, accessible by the Post-Processing Unit. In the forthcoming sections, each of these modules is described in more detail.

4.1 Processing System (PS)

Our design is based on well-defined interfaces and minimal dependencies across the modules, enabling a high degree of decoupling between them. For this reason, a generic interface should be provided to user programs to interact with our architecture. This interface utilizes a standardized PS/PL interaction protocol, offering support for various CPUs and software, ranging from bare-metal programs to full Linux operating systems.

The interface allows a set of instructions $I \in \{i_0, ..., i_n\}$ to be sent from the PS to the PL and a set of corresponding results $Res \in \{res_0, ..., res_n\}$, responding from the PL based on the previously sent commands. In addition, each instruction $i := \{i_o, i_{pyl}\}$ consists of an instruction opcode i_o recognizing the instruction type and a payload i_{pyl} specifying the parameters required by the corresponding instruction. In this initial implementation, four instructions are supported:

- *save_secret:* awaits a payload $i_{pyl} := \{k_I, v_I\}$ to store the initialization secret (k_I, v_I) comprising a secret cryptographic key k_I and a randomly chosen and public initialization vector v_I on the PL in volatile memory.
- *retrieve_puf*: accepts a payload $i_{pyl} := \{s_{puf}, d_I, c, m_r\}$ consisting of the type of PUF s_{puf}, the initialization data $d_I \in \{0,1\}^{m_d}$ to initialize the selected PUF cells of data bus width m_d, and PUF challenge c depending on s_{puf}. Finally, the bit size of the response is specified by m_r. After parsing the payload, a set of randomly selected memory addresses $A_{puf} = \{a_0, ..., a_{(m_r/m_d)-1}\}$ is derived from (k_I, v_I). The cells in A_{puf} are initialized with d_I, and the PUF of type s_{puf} is executed with a challenge c, as further described in Section 4.3.1. The execution results in m_r response bits $r_{raw} \in \{0,1\}^{m_r}$. This response is corrected to k_{puf} utilizing helper data *hd*. Finally, k_{puf} is stored in the PL. A description of the different steps of this instruction is given in Equation (1). It shows the random address generation, the PUF readout, and the PUF post-processing, eliminating instabilities and improving the entropy of the raw PUF response r_{raw}.

$$\begin{aligned} A_{puf} &:= pau(k_I, v_I) \\ r_{raw} &:= puf(s_{puf}, A_{puf}, d_I, c) \\ k_{puf} &:= postproc_{hd}(r_{raw}) \end{aligned} \tag{1}$$

- *write_data*: loads the user key k_I and the corrected PUF response k_{puf}, as well as the initialization vector v_I from the Storage Controller. These keys are used to encrypt the data received from the memory connected to the PL. To store v_I and k_{puf}, a previous execution of save_secret and retrieve_puf is necessary. The payload $i_{pyl} := \{\mathcal{P}, a\}$ consists of the write address a and the plaintext $\mathcal{P}$ to be encrypted. As described in Equation (2), this instruction encrypts $\mathcal{P}$ and writes the resulting ciphertext C to the cells identified by address a.

$$\begin{aligned} C &:= encrypt_{k_{puf}, k_I}(\mathcal{P}, a) \\ res &:= write(C, a) \end{aligned} \tag{2}$$

- *read_data*: Similar to write_data, two keys denoted as k_I and k_{puf} are required to decrypt data requested from the memory module. As described in Equation (3), this instruction accepts an address a as payload $i_{pyl} := \{a\}$, necessary to read the corresponding data from the memory module and subsequently execute the decryption operation. Finally, the plaintext $\mathcal{P}$ is returned to the PS

$$\begin{aligned} C &:= read(a) \\ \mathcal{P} &:= decrypt_{k_{puf},k_I}(C, a) \end{aligned} \tag{3}$$

Upon executing an instruction i_j, a corresponding result $res_j := \{i_o, i_{state}, \mathcal{P}\}$ is returned from the PL and forwarded to the user application executed on the PS. The result includes the state $i_{state} \in \{success, failed\}$ of the previously executed instruction identified by opcode i_o, together with the identifier of i_j. In the case of an i_j with $i_o := read\_data$, the decrypted value $\mathcal{P}$ is returned along with i_{state}.

4.2 PS/PL Interconnection

On the PL, the PS/PL interconnection module is responsible for receiving instructions from the CPU and responding with corresponding results. This module should provide a First In – First Out (FIFO)-storage mechanism, buffering multiple instructions i and forwarding them to the Instruction Decoder. It additionally waits for the corresponding results res_j from the Instruction Decoder. If a result is received, it is forwarded to the PS, and the corresponding i_j is removed from the FIFO. To avoid unreliable behavior in the later implemented prototype, a command i_{j+1} is only executed after receiving the result res_j of the previous instruction i_j. To enhance the overall throughput, the whole architecture could be designed as a pipeline. However, for the sake of simplicity and to avoid the increased area consumption caused by implementing additional buffers, this approach is not adopted in this work.

4.3 Programming Logic (PL)

This section describes the different modules implemented on the PL, constituting the core functionality of our data-binding architecture.

4.3.1 PUF Controller. This module is responsible for conducting PUF measurements of type s_{puf}. It acquires a set of randomly selected memory addresses from the PUF Allocation Unit, as outlined in the subsequent section. Subsequently, it delegates the PUF-dependent read and write operations to the memory controller. Finally, this module handles the transmission of the PUF response r_{raw} to the Post-Processing Unit, which transforms them into a cryptographically usable key k_{puf}. The actual PUF implementation is described in Section 5.0.3.

4.3.2 PUF Allocation Unit (PAU). In order to increase the resistance against an attacker capable of determining the memory cells queried by the PUF implementation, additional measures have to be taken. To this end, we have implemented a novel method that generates PUF responses based on a selection of random addresses for PUF readout. This method enhances the resilience against hardware attacks. The PUF Allocation Unit (PAU) provides functionality for creating a set of randomly distributed addresses within the valid address space of the connected memory module. The generation is based on the initialization secret (k_I, v_I) and utilizes the random numbers generated through Equation (5). Depending on the block size of the cipher, these numbers then may have to be mapped to the space of valid addresses using some function $\varphi\{0,1\}^{m_b} \rightarrow \{0,1\}^{m_a}$. Formally, the composition of both steps is defined through Equation (4).

$$pau(k_I, v_I) = (\varphi \circ gen\_rand_{k_{puf},k_I})\{0, 1, \dots\} \tag{4}$$

4.3.3 Crypto Core. In our architecture, encryption is provided by a block cipher, formally defined through the function $E_k(\mathcal{P}) = C$. The Crypto Core extends it by two additional modes of operation. Counter Mode (CTR) mode is used by the PAU with $k = k_I$ and $IV = v_I$ for the generation of random values through Equation (5).

$$gen\_rand_{k_{puf},k_I}(ctr) = E_{k_I}(v_I + ctr) \tag{5}$$

Further, the well-established Xor–Encrypt–Xor (XEX) mode [37] is used for the encryption and decryption of data to be stored in memory. As shown in Equation (6), we generate a tweak T_a through encryption of the address a of the memory cell that is written to or read from, which is then XORed with the respective data. For both operations, a common key $k = k_{puf} \oplus k_I$ is used. We do not split the memory into sectors, and thus, we multiply each tweak T by a constant value of 2.

$$\begin{aligned} gen\_tweak_{k_{puf},k_I}(a) &= 2E_{k_{puf} \oplus k_I}(a) \\ encrypt_{k_{puf},k_I}(\mathcal{P}, a) &= E_{k_{puf} \oplus k_I}(T_a \oplus \mathcal{P}) \oplus T_a \\ decrypt_{k_{puf},k_I}(C, a) &= E^{-1}_{k_{puf} \oplus k_I}(T_a \oplus C) \oplus T_a \end{aligned} \tag{6}$$

4.3.4 Memory Controller. To perform the physical read and write operations initiated by the *write_data* and *read_data* instructions or to implement various PUF-types on externally connected memory modules, a custom memory controller is required. This controller offers an interface, accepting an address a of width m_a in order to read the value at the designated address or an address a and data value d of width m_d to write data. m_d must be equal to or a multiple of the data bus width of the connected memory. If the data bus width of the memory module is smaller than m_d, multiple read and write operations are performed. To avoid an additional mapping of virtual addresses to logical addresses, we define the address width m_a to be equal to the address width of the memory module in this prototype implementation.

4.3.5 Instruction Decoder. This module awaits an instruction $i := \{i_o, i_{pyl}\}$, transmitted by the PS/PL Interconnection Module. Upon reception, i_{pyl} is decomposed based on the opcode i_o, and the decomposed values are forwarded to the required modules to execute the instruction. Furthermore, the Instruction Decoder handles the initialization of the modules and maintains their order of execution. Ultimately, the modules respond to the Instruction Decoder after the instructions have been executed. Finally, a result res is generated and forwarded to the PS/PL Interconnection Module.

4.3.6 Post-Processing Unit. This module is responsible for the correction of errors occurring in a raw PUF response r_{raw} with the aid of helper data *hd* resulting in a cryptographically usable key

k_{puf}. Otherwise, unstable responses would lead to minor variations in cryptographic keys, preventing the successful decryption of stored data. The required helper data is generated from a series of PUF measurements conducted from memory blocks selected by the PAU. This process is performed during the enrollment phase. All measurements are carried out under the same challenge c and subsequently stored on non-volatile memory accessible from the FPGA. The generation of helper data hd exclusively for this selection of cells enables a comparable small helper data size. Furthermore, in scenarios requiring key revocation or increased instability due to aging effects, the enrollment process can be repeated. This repetition ensures the creation of a fresh set of helper data, thereby preserving system security and stability. We propose helper data schemes that operate on blocks of size k. The size of the necessary helper data hd depends on the number of selected cells, the chosen error correction algorithm, its message length k, and code word length n, and thus the resulting code rate $R = n/k$.

5 IMPLEMENTATION

In this section, we elaborate on the implementation based on the previously introduced conceptual design. All modules defined in Section 4 are implemented as single IP-core or, in the case of the memory controller, subdivided into multiple IP-cores. This allows for a high degree of decoupling, simplifies the extension of additional modules added by the user, and improves the overall scalability. In addition to the module-specific inputs and outputs, each module provides a generic interface to start the module operation and to indicate its state. This allows for a seamless synchronization between the modules, driven by a single clock source.

5.0.1 Processing System. To demonstrate the interaction with user applications, we implemented two program variants on the PS. The first one is implemented as a bare-metal program using the xaxidma driver provided by Xilinx. We could demonstrate the support for Direct Memory Access (DMA)-polling mode as well as the implementation using interrupts. Furthermore, we have implemented a kernel that supports the same interface accessible from a PetaLinux operating system.

5.0.2 PS/PL Interconnection. A high-performance Advanced eXtensible Interface (AXI) interface implements the communication between the PS and PL. We have implemented an AXI-stream interface, which supports high bandwidths and the DMA mapping into the address space of the PL [6]. The AXI interface, a custom FIFO module, as well as the DMA functionality utilizing the AXI-DMA IP-core [3] is combined in the PS/PL Interconnection IP-core and connected to the PS. The block design of the IP-core is depicted in Figure 2. It provides interfaces to send instructions and receive results from the Instruction Decoder, established by an AXI-slave and AXI-master interface. These two interfaces are connected via AXI interconnect modules to the DMA controller and a custom FIFO implementation.

5.0.3 PUF Controller. This module is responsible for the execution of PUFs on external Non-Volatile Memories (NVMs). In this prototype implementation, the PUF types $s_{puf} \in \{read\_latency, write\_latency\}$ are supported, similar to the implementation denoted in the works of Khan *et al.* [25] or Zhang *et al.* [52]. The

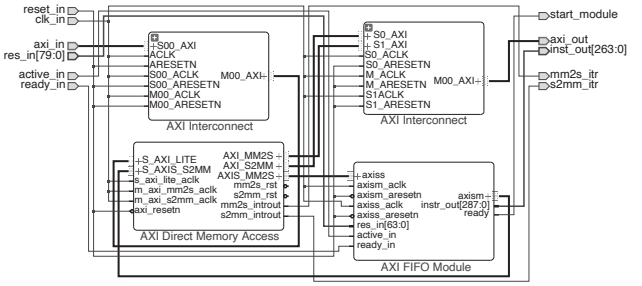

Figure 2: Module showing the interconnection module supporting the AXI-streaming interface to transmit data between the PL and PS.

foundation of these PUF implementations lies in the intentional violation of the timing of the memory module, for example, by reading or writing in faster intervals than specified in the manual of the memory chip or setting certain signals of the communication protocol for shorter periods of time, e.g., the Chip Enable ($\overline{CE}$) or Write Enable ($\overline{WE}$) signals when communicating with SRAM protocol-compatible memory modules. These violations result in bit-flips, which are used as PUF response r_{raw} and, after applying post-processing, as cryptographic key k_{puf}. Furthermore, it is possible to extend the implementation by PUF implementations on other NVMs, like PUFs based on Phase Change Memory (PCM) [51], Resistive Random Access Memory (ReRAM) [43], or well-established flash memory [4, 35, 39, 44].

To start the PUF execution, the initial tuple (k_I, v_I) stored on the PL is forwarded to the PAU to generate a random selection of (m_r/m_d) memory cell addresses A_{puf}. In the first step, the generation of a backup of the currently stored values requested at the addresses A_{puf} is necessary. The contents of the memory cells are stored in registers on the FPGA and immediately restored directly after each write operation. Afterwards, the PUF Controller initializes the data cells addressed by A_{puf} with d_I, using the timing specifications as defined in the data-sheet of the connected memory. Subsequently, the PUF measurement of type s_{puf} is performed, based on a challenge c, resulting in a raw response r_{raw}, which is forwarded to the Post-Processing Unit to generate a cryptographically usable key k_{puf}. In the case of latency PUFs, the challenge c defines the timing reduction of the write cycle time or read cycle time applied during PUF execution. Finally, the backed-up value is restored, avoiding the need to reserve a dedicated memory area for exclusive PUF use.

5.0.4 PUF Allocation Unit. This module utilizes the CTR-mode offered by the Crypto Core as described in the subsequent Section 5.0.5, for the generation of m_b-bit wide random values, depicted as *PUF Addr Gen* in Figure 3. Every resulting block is then converted into (m_b/m_a) valid addresses by simply splitting it into slices of size m_a bits. The remaining bits are discarded. Note that only in the case of $m_a = m_b$, no address collisions occur. Otherwise, the probability of collisions has to be evaluated, and m_r incremented accordingly to provide a sufficient amount of unique addresses.

5.0.5 Crypto Core. To achieve a high level of security while simultaneously minimizing area consumption and latency, we employ

and evaluate a set of well-established lightweight ciphers, namely SIMON [10], PRINCE [13] and PRINCEv2 [15]. As a baseline, we compare those to an existing implementation of the AES algorithm [42], where all of these operate with 128-bit keys. To evaluate the trade-offs between total latency, maximum clock frequency, and area consumption, we implemented optimized round-based variants as well as partly and fully unrolled versions of the lightweight ciphers. To further minimize the area impact caused by the cryptographic cipher, a single block cipher instance is shared among the components of the design as depicted in Figure 3. The instance is wrapped within a Crypto Core module, which offers both CTR- and XEX-mode and, thus, covers all cryptographic use cases described in Section 4.3.3.

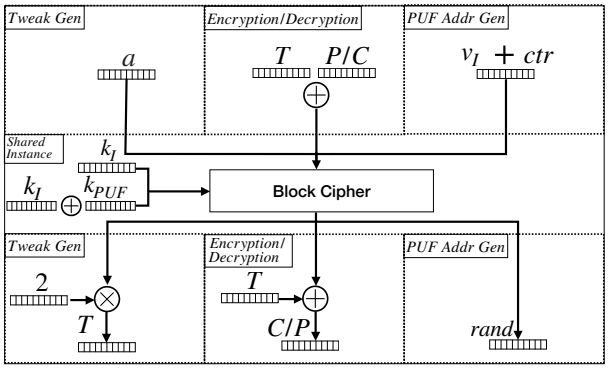

Figure 3: Reusing a single instance of a block cipher to generate random PUF addresses, to encrypt data in XEX-mode and to derive the encryption key.

5.0.6 Memory Controller. In our proof-of-concept implementation, a memory controller supporting the SRAM-protocol was developed, enabling seamless interaction not only with SRAM memories but also a variety of other memory modules, including Magnetoresistive Random Access Memory (MRAM) and Ferroelectric Random Access Memory (FRAM) modules. Similar to all other components in our framework, this component can simply be exchanged to support other transmission protocols, for example, Open NAND Flash Interface (ONFI) or Serial Peripheral Interface (SPI). To enable the readout of latency PUFs and the interaction with a variety of different memory modules, the timing parameters of the communication protocol are adjustable during runtime. To achieve a precise timing resolution, the memory controller is driven by higher clock frequencies compared to the rest of the architecture. This can be achieved by Xilinx clocking Wizard IP-core [7] that uses the frequency of the base clock source to generate higher clock frequencies.

To leverage the high clock frequencies, the module is implemented as a state machine with very short critical paths, controlling the $\overline{WE}$, Output Enable ($\overline{OE}$), and $\overline{CE}$ signals, as well as the data bus (d) and address bus (a) wires, according to the standard SRAM-protocol. The Memory Controller module is subdivided into two IP-cores: one implementing the write and the other one implementing the read operation. For our prototype implementation, the timing parameters of a full write t_{wc} and read cycle t_{rc} can be adjusted during runtime. Specifically, the t_{pwc}, which defines the time when both $\overline{WE}$ and $\overline{CE}$ are active. This is necessary to execute write-latency PUFs. To execute read-latency PUFs, the t_{prc} defining when $\overline{OE}$ and $\overline{CE}$ are active can be adjusted dynamically. This allows for a PUF construction similar to the implementation described in [25]. The memory modules under test were connected to the logical pins of the FPGA, defined in a constraints file. The connection is established by a custom-made Printed Circuit Board (PCB).

5.0.7 Instruction Decoder. This module is implemented as a four-ary state machine, similar to the structure of CPU-pipelines. In the first phase (*fetch*), the module waits for an instruction i from the PS/PL Interconnection module. Based on the opcode i_o, the payload i_{pyl} is parsed and forwarded to the dedicated modules. We refer to this phase as the *decode*-phase. If all signals are set, the required modules are started by triggering their start signals while maintaining the timing requirements and synchronization between the modules. This is done during the *execution*-phase. Afterward, the Instruction Decoder waits for all modules to complete their operations before sending the result res to the PS/PL Interconnection Module in a phase, which is referred to as *write-back*.

5.0.8 Post-Processing Unit. The design of our architecture allows for an adaption of various error correction algorithms. Instead of implementing a specific algorithm in this work, a generic module was implemented, which allows the adoption of a wide range of error correction algorithms, as outlined in the existing literature. For example, an implementation provided by Bösch *et al.* presenting two efficient PUF post-processing algorithms implemented on FPGAs utilizing the well-known Reed-Muller and Golay codes [14]. Reed-Muller codes allow a correction of 1 out of 3 error bits when using the parameters $k = 1$ and $n = 3$. Due to its high code rate $R = 3$, a large amount of helper data is necessary. Goley codes allow only the correction of 3 bits within $n = 24$-bit blocks. A further implementation implementing Bose-Chaudhuri-Hocquenghem-Codes (BCH) and convolutional codes for PUF post-processing on FPGAs was presented by Jarvis *et al.* [24]. BCH codes are capable of correcting 18 errors in each 255-bit block and produce helper data with a rate of 1.95 to the input data. All these implementations are capable of correcting PUF errors arising from intrinsic memory-based PUFs such as those utilized in this work. A further selection of different PUF post-processing algorithms implemented on FPGAs was conducted by Hiller *et al.* [23]. In their work, different linear and pointer-based post-processing schemes are compared and evaluated on Xilinx Spartan 6, Zync 7020, and Ultrascale boards. Consequently, these implementations seamlessly integrate into our data-binding architecture, facilitating the correction of prominent PUF-related errors.

6 PERFORMANCE EVALUATION

The efficiency of our implementation is assessed through a comprehensive evaluation of the overall architecture shown in Figure 1. Therefore, we evaluate the area consumption required to implement our design, as well as the latency of the critical paths, the derived maximum clock frequency, and the number of clock cycles per instruction.

6.0.1 Test setup. Considering the prospective implementation of our architecture in mobile telecommunication networks, as described in Section 8, showcasing our implementation's adaptability to resource-constrained and high-performance environments, we

have conducted an assessment of compatible hardware. The selection of hardware to evaluate is based on the manuscript published by Xilinx in which the eligibility of FPGAs in wireless base station connectivity is discussed [33]. As presented in this work, Xilinx's Series 7 FPGAs exhibits a high level of throughput and flexibility when implementing standard network communication interfaces. This is attributed to their highly parallel execution and a significant quantity of high-speed transceivers and I/O ports. Xilinx's Series 7 provides three distinct FPGA families optimized for different use cases: Artix™ 7 (Optimized for low cost, low package size and power consumption), Kintex ™ 7 (Balanced between power-consumption, latency and capacity), and Virtex™ 7 (Optimized for highest performance and capacity) [2]. To demonstrate the high adaptability to different types of devices, we perform benchmark tests on boards of all three categories as shown in Table 1. Specifically, the Kintex™ 7 XC7K355T model evaluated in this paper features 24 serial transceivers capable of achieving speeds of up to 12.5 Gbit/s, making this model suitable for low-end networking applications. The paper published by Xilinx further utilizes the XC7VX415T, XC7VX690T, and XC7VX980T models, which respectively accommodate 48, 80, and 72 transceivers while supporting data rates of up to 13.1 Gbit/s. From this list, we consider Virtex™ 7 XC7VX415T and XC7VX980T models in our assessment.

An evaluation of the architecture utilizing ARM CPUs is done through the resource-constrained PYNQ-Z2 board and the Zync® UltraScale+™ MPSoC based on a Zync® UltraScale+™ ZCU102 board, which supports the execution of bare-metal programs as well as full PetaLinux kernels. In addition to the above-mentioned physical ARM CPU cores, our implementation is also evaluated incorporating Microblaze™ soft-cores to support devices not containing a physical CPU.

The benchmark results in the subsequent section were obtained from a prototype implementation that leverages the AXI-streaming interface, configured with a beat size of 32 bit, a maximum burst size of 16 beats, a data width $m_d := 64$ bit, equal to the block size of the used PRINCE cipher as well as an address width of $m_a := 15$ bit, which is equal to the address width of the connected memory module. The FIFO within the PL/PS Interaction Module reserves space for commands and answers, each with a capacity of ten. Commands can reach lengths of up to 264 bit when considering the longest instruction *save_secret* consisting of a $i_o := 8$ bit opcode the key k_I and v_I each with a length of 128 bit. Furthermore, ciphers with a block size of 64 bit and a key size $m_k := 128$ bit are utilized. Our prototype is connected to a Lapis MR48V256C FRAM memory module supporting the SRAM communication protocol with $t_{pwc} = t_{prc} := 150$ ns and a write-latency PUF implementation with $t_{thresh} := 55$ ns. The memory controller is adjusted to align with the memory's specification, including a data width of 8 bit and an address width of 15 bit. To match with m_d, each read and write operation requires eight accesses by the memory controller. Subsequently, the Storage Controller supports data widths of 16 bit.

6.0.2 Area consumption. An important property of our design is the number of Lookup-tables (LUTs) and Flip-Flops (FFs) utilized when implementing our design on different FPGAs. A small number of LUTs and FFs allows for the adoption of our design on devices offering different resources in terms of area. Additionally, a low area consumption leaves more space to implement further logic for an application integrated with the data-binding architecture. Figure 4a shows the relative number of LUTs and FFs consumed per module on each of the devices listed in Table 1. Notably, the PS/PL Interconnection Module occupies the largest area across all modules, probably caused by the overhead of the generic AXI-bus, the DMA-module, such as the instruction and answer FIFO. Furthermore, the PUF Control Unit occupies only 121 LUTs and 242 FFs in total, independent of the length of the PUF-response, as long as no adjustment of the data- and address-bus is necessary. In comparison, the PUF implementations utilized in the architectures presented in Section 2 occupy higher amounts of area. The PUF used by Sepulveda *et al.* [41] occupies 1288 LUTs and 945 FFs when implementing 128 ROs and the PUF-implementation utilized by Zhang *et al.* [49] occupies about 1400 LUTs to generate 128-bit responses [5].

Table 1: Devices used for our performance evaluation [2].

FPGA Part/Board	Series	# LUTS	# FFs
PYNQ-Z2	Zynq®-7000 Artix™ 7	53200	106400
XC7K355T	Kintex™ 7	222600	445200
XC7VX415T	Virtex™ 7	257600	515200
Ultra-Scale+ ZCU102	UltraScale+™	274080	548160
XC7VX980T	Virtex™ 7	612000	1224000

To estimate the footprint of the whole architecture, we further analyze different post-processing algorithms required to transform r_{raw} to k_{puf}. Therefore, Bösch *et al.* [14] presents efficient implementations of fuzzy extractors on FPGAs, including the repetition and Reed-Muller code, where both occupy less than 600 LUTs. Jarvis *et al.* implemented a suitable decoder of a BCH code with about 870 LUTs. This implementation occupies only 0.81% additional space on a resource-constraint AMD PYNQ-Z2. Moreover, Hiller et al. [23] employed various post-processing algorithms in their conducted analyses. These algorithms occupy an area of 43 to 243 slices, which would allow a seamless integration into our existing architecture. We presume that the necessary helper data is stored in non-volatile memory accessible to the FPGA. Consequently, the storage of helper data is not included in the count of LUTs and FFs.

In addition to the Post-Processing Unit, the performance of our design is heavily reliant on the cryptographic cipher utilized in our design. Hence, we have evaluated the impact on a set of different ciphers, as explained in Section 5.0.5. The area consumption of different hardware-implemented ciphers across the devices listed in Table 1 is depicted in Figure 4b.

The fully unrolled version of SIMON occupies the most space, requiring 6867 LUTs to unroll each of the 44 rounds. Comparatively, among the round-based variants, the AES implementation has the largest area consumption, utilizing 3327 LUTs and 2990 FFs. In contrast, the round-based version of PRINCE exhibits 541 LUTs and 259 FFs, utilizing the smallest area and additionally less than half of the space of its unrolled variant. The difference in the area consumption when comparing the unrolled versions of PRINCE and PRINCEv2 is negligible. Our partially unrolled version of SIMON always unrolling four rounds occupies 3088 LUTs and 132 FFs.

6.0.3 Latency. To allow for a seamless synchronization between the different modules, our FPGA-based data-binding architecture is driven by a single clock source. Only the memory controller

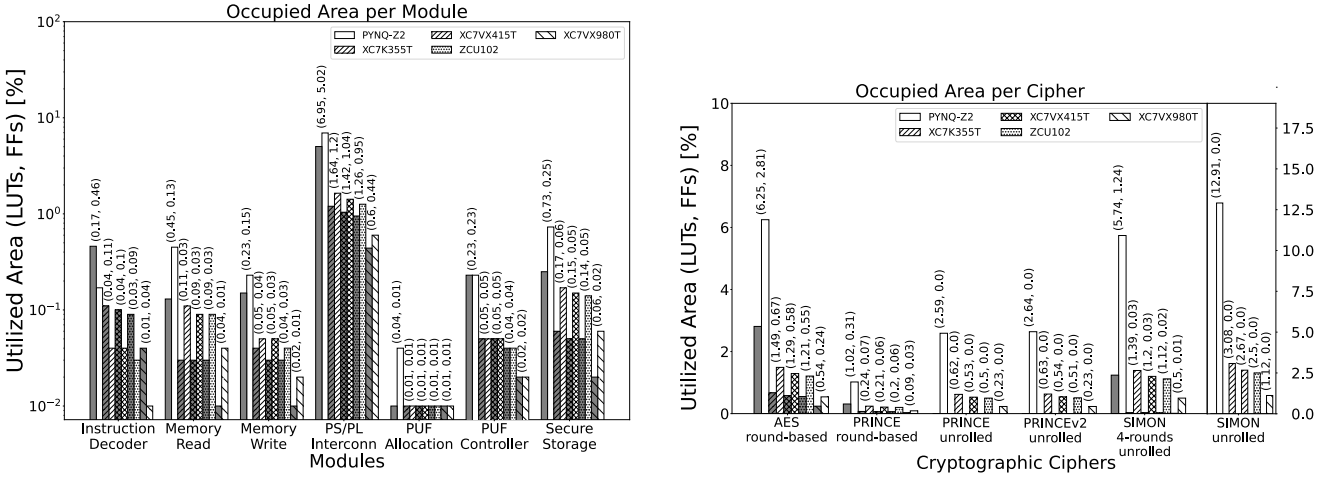

Figure 4: Relative number of LUTs (white background color) and FFs (grey background color) occupied by each of the modules of the implementation evaluated on the devices of Table 1.

requires higher clock frequencies of 400 MHz to match the timing requirements of the connected memory module and allow for a precise PUF readout. In general, the overall latency of each instruction i can be described by the frequency of the clock driving the architecture, as well as the number of clock cycles per instruction. As a result, it is essential to examine the maximum clock frequency capable of driving all modules, followed by an analysis of the number of clock cycles required by each module and instruction. Determining the maximum clock frequency requires a calculation of the latency of the critical path of our implementation. This can be accomplished using Xilinx Vivado 2022.1, capable of determining the Worst Negative Slack (WNS) on each device denoted in Table 1 separately.

The WNS denotes the remaining time within one clock cycle when considering the latency of the critical path t_{cp}, which can be determined by calculating $t_{cp} = T_{clk} - t_{wns}$ where T_{clk} is the period of one clock cycle and t_{wns} the determined WNS. The evaluated WNS values of all modules show that regardless of the employed cryptographic cipher, the critical path in our design lies always within the implementation of the cipher. In order to determine the maximum clock frequency of our design, the WNS of each cryptographic module is examined and used to derive the delay of the critical path t_{cp}. Subsequently, t_{cp} is utilized to calculate the total delay of a single encryption operation t_{tot} based on the number of cycles #cc required to encrypt a single block. Furthermore, t_{cp} is used to derive the maximum frequency f_{max} as described above.

Table 2 provides an overview of the above-discussed timing values evaluated for each cipher. In general, it can be seen that unrolling the designs of PRINCE and SIMON results in a significant reduction of t_{tot} of their round-based version due to execution within a single clock cycle. Conversely, the unrolled versions suffer from much longer critical paths, resulting in higher t_{cp} and decreasing the maximum frequency f_{max} of the whole design. Hence, we consider the round-based version of PRINCE for our prototype implementation, which requires 10 cc instead of one but allows executing the whole design with frequencies of up to 134 MHz on the resource-constrained PYNQ-Z2 and with even higher frequencies on the other boards under evaluation. This is a significant enhancement compared to 32 MHz when executing the unrolled version on the PYNQ-Z2. After determining f_{max} of our design, we measured the latency of the overall architecture by evaluating the #cc of each instruction i through the modules m independently. The individual cc per module and instruction are displayed in Table 3.

As described in Section 6.0.1, instructions are encoded by 264-bit vectors to be able to store the longest command *save_secret*, causing a constant overhead of 10 cc to transmit an AXI burst of 9 beats each consisting of 32 bit. The Instruction Decoder requires one clock cycle for each state of its four-ary state machine, consisting of a fetch, decode, execute, and write-back phase. The storage controller requires 8 cc to transmit a 128-bit k_I, v_I or k_{puf} when considering a 16-bit data bus to the Storage Controller. Consequently, during the execution of *save_secret*, the tuple $|(k_I, v_I)| \coloneqq 256$ bit must be stored within the module, whereas when reading and writing data k_I, v_I, and k_{puf} are loaded from the module and when executing $retrieve\_puf$, k_{puf} is stored and k_I and v_I are loaded to run the PUF Allocation Unit. The XEX mode requires the execution of the cipher twice per read and write operation, therefore, resulting in 21 cc (20 cc for cipher execution and 1 cc for module synchronization). The memory controller in our implementation operates on a 400 MHz clock to achieve the high precision necessary to execute the PUF and to meet the timing specification of the memory module. These modules are synchronized with the other modules driven by a 100 MHz clock after the read and write operations. Therefore, we track the number of clock cycles the 100 MHz clock waits for the memory controller to complete its tasks. Assuming an ideal PUF without the need for post-processing, extraction of data for a 128-bit key takes 1048 cc. This is determined by the number of memory accesses with respect to a 8-bit data width and the read

Table 2: Timing parameters of each cipher, evaluated on various FPGAs. t_{cp} is depicted in ns and f_{max} in MHz. The round-based version of PRINCE is selected in our architecture.

	PYNQ-Z2				XC7K355T				XC7VX415T				XC7VX980T				UltraScale+ ZCU102			
	t_{cp}	#cc	t_{tot}	f_{max}	t_{cp}	#cc	t_{tot}	f_{max}	t_{cp}	#cc	t_{tot}	f_{max}	t_{cp}	#cc	t_{tot}	f_{max}	t_{cp}	#cc	t_{tot}	f_{max}
AES round-based	8.86	46	407.7	112	4.74	46	218.2	210	4.00	46	183.8	250	4.80	46	220.3	208	2.78	46	127.7	360
PRINCE round-based	7.41	10	74.1	134	3.49	10	34.9	286	3.42	10	34.2	292	3.55	10	35.5	282	2.43	10	24.3	412
PRINCE unrolled	30.62	1	30.6	32	15.2	1	15.2	65	15.44	1	15.4	64	15.2	1	15.2	65	7.9	1	7.9	126
PRINCEv2 unrolled	29.71	1	29.7	33	7.96	1	8.0	125	15.93	1	15.9	62	15.5	1	15.5	64	7.9	1	7.9	126
SIMON 4 rounds unrolled	7.00	11	77.0	143	3.45	11	38.0	264	3.68	11	40.5	272	3.80	11	41.8	239	2.23	11	24.5	448
SIMON unrolled	69.42	1	69.4	15	39.59	1	39.6	26	40.51	1	40.5	25	39.83	1	39.8	26	26.39	1	26.4	38

Table 3: Clock cycles (*cc*) and time consumption per module driven by a 100 MHz clock.

	Instruction *i*			
Module	**save_secret**	**retrieve_puf**	**write_data**	**read_data**
PS/PL Interconnection	10 *cc*	10 *cc*	10 *cc*	10 *cc*
Instruction Decoder	4 *cc*	4 *cc*	4 *cc*	4 *cc*
Storage Controller	16 *cc*	24 *cc*	24 *cc*	24 *cc*
Crypto Core	/	161 *cc*	21 *cc*	21 *cc*
PUF Controller	/	69 *cc*	/	/
PUF Allocation Unit	/	16 *cc*	/	/
Memory Controller	/	1048 *cc*	18 *cc*	18 *cc*
Total cycles	30 *cc*	1332 *cc*	77 *cc*	77 *cc*
Time Consumption	0.30 µs	13.32 µs	0.77 µs	0.77 µs

and write latency of the memory controller for querying and restoring the affected memory cells. In reality, PUF responses are noisy, and often, more than 4000 bits of data are required, along with the execution of error-correcting codes. This increases the delay for querying the required data to around $33k$ *cc* or 330 µs. Using the approach by Jarvis *et al.* [24], around $51k$ *cc* or 490 µs are required for decoding with negligible failure probability and sufficient left-over entropy. While these values are high, it is to be noted that this latency does not occur during run-time but only once at the start-up of the device. Therefore, this does not affect the delay of a read or write operation. For a read and write operation, 17 *cc* are needed to satisfy the timing requirement with $t_{rc} = t_{wc} := 150$ ns of the FRAM module under test. When executing read or write requests, it takes one clock cycle to synchronize either with the PUF Controller or the Instruction Decoder. Finally, a read or write request requires 77 *cc* or 0.77 µs based on a period of $T_{clk} := 10$ ns. When considering $f_{max} := 134$ MHz, as evaluated in Table 2, a read and write requests would be possible in 0.58 µs; on the Zync® UltraScale+™ ZCU102, read and write operations could be executed within 0.19 µs using a frequency of $f_{max} := 412$ MHz. It is important to note that these values do not consider effects like clock skews and jitters.

The previously described assessment focuses only on the latency introduced within the PL. An additional assessment of the PS is essential to evaluate the entire system. Furthermore, our investigation includes the analysis of the delays caused by the PS/PL interconnection. To facilitate these measurements, the PS/PL Interconnection Module was tested by a distinct block design and connected to a custom FIFO module, which orchestrates the data loop-back, allowing for exclusive measurements of the latency of the PS/PL interface. To conduct these measurements, we implemented a small bare-metal program using Xilinx's xaxidma library to measure the loop-back duration while transmitting 32 B data blocks. The validity of the sent data is additionally analyzed using the System Integrated Logic Analyzer (ILA) IP-core [1]. These measurements are executed on the PYNQ-Z2 and ZCU102 boards, with the program running on the first core of its ARM® Cortex™-A9 CPU and on a single core of its ARM® Cortex™-A53, as well as on its Cortex™-R5F and a Microblaze™ soft-core. The results obtained from the bare-metal program, conducted over 1000 iterations, are illustrated in Figure 5.

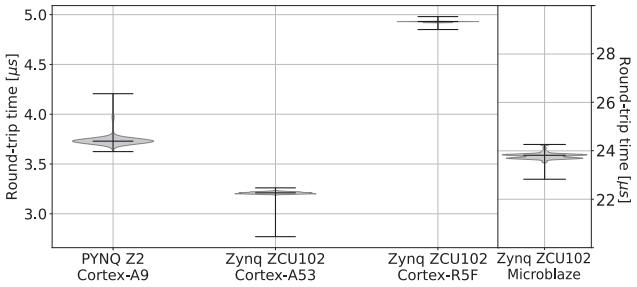

Figure 5: Round-trip time when executing the loop-back test, evaluated across 1000 iterations.

It can be seen that the overhead caused by the transmission and reception of 32 B messages leads to a maximum latency of 4.21 µs on the Cortex™-A9, 3.26 µs on the Cortex™-A53 and 4.98 µs on the Cortex™-R5F. Hence, the PS/PL Interconnection is responsible for a significant portion of the total latency when driving the AXI bus with a 100 MHz clock. In general, even when running code on the Cortex™-R5F real-time CPU with 500 MHz, read and write instructions on the FRAM memory module require about 4.5 µs. It should be noted that the lightweight cryptographic cipher consumes only 0.21 µs the overall latency when executing read or write instructions. In the end, we evaluated the additional time consumption caused by a user space application using a Linux kernel module to send data over the AXI-streaming interface, running on the PYNQ-Z2 incorporating a Cortex™-A9. The time consumption of 1000 bidirectional requests can be seen in Figure 6. We notice an average delay centered around 60 µs, which seems to follow the observation as discussed in [21].

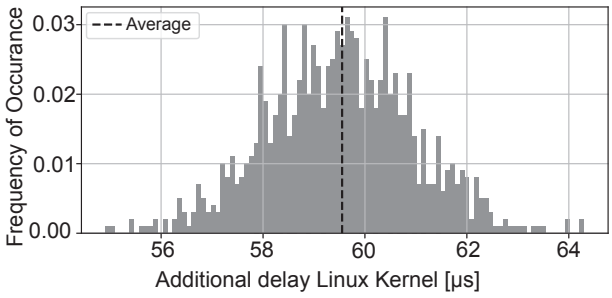

Figure 6: Histogram showing the additional latency caused by the Linux Kernel and the implemented kernel module in contrast to the bare-metal implementation.

7 SECURITY EVALUATION

Next, we assess the confidentiality of data that is encrypted and stored on our device against the abilities of a computationally bound attacker presented in Section 3. The first three abilities of $\mathcal{A}$ require an analysis against a Chosen-Ciphertext Attack (CCA). In their paper, Minematsu determines that if the underlying n-bit block cipher is secure, then XEX offers only negligibly small CCA-advantage to $\mathcal{A}$, as long as the number of processed blocks is sufficiently smaller than $2^{n/2-1}$ [32]. We conclude that if this bound is satisfied by choosing the block size m_b large enough with respect to the size of the address space m_a, our construction is CCA-secure. It is to note that, in the same paper, Minematsu presented a weakness of XEX. However, this weakness has no impact on our design, as we do not split the memory module into sectors and set $\alpha = 2$ to a constant value.

The last two abilities of $\mathcal{A}$ consider the consequences of physical access to the device. Through possession of the memory module, $\mathcal{A}$ can recover k_{puf} through evaluation of the PUF on chosen parameters. To successfully recover the key, knowledge of the value and order of the (m_r/m_d) PUF responses is necessary. Under the assumption of an ideal PUF, when choosing the number of addressable memory cells m_a sufficiently large, the probability of $\mathcal{A}$ being able to single out the cells used for the generation of k_{puf} becomes negligible. However, depending on the uniqueness of the PUF in place, the responses of the memory cells might correlate, decreasing both the entropy in the PUF response and the size of the search space. To combat this, the amount of queried data (m_r/m_d) must be chosen large enough to provide sufficient entropy, and the number of memory cells with unique responses with respect to the PUF must be determined to calculate the security margin. Moreover, by employing our random PUF cell selection, based on an inaccessible parameter k_I, modeling attacks on PUFs can be mitigated. The random choice of cells within a sufficiently large memory module generates a broad set of uncorrelated challenge-response pairs, which makes machine learning attacks as described in [38] inapplicable. Effects within the PUF, such as the spatial correlation between the memory cells, must be specifically analyzed for the employed PUF [45]. In general, we assume a PUF to be resistant to both mathematical and physical cloning. As the key is only stored within the device and in volatile memory and re-generated after powering on the device based on the externally provided secret (k_I, v_I), theft of the device does not leak k_{puf} and, thus, provides no advantage over the first three abilities of $\mathcal{A}$.

Current cryptanalysis indicates only attacks on a reduced number of rounds when utilizing the well-studied ciphers PRINCE [19][34] and SIMON [11] offered by our implementation. To support more lightweight implementations, we combine k_{puf} with the externally provided k_I. We assume that k_I, such as k_{puf} provides a sufficient degree of entropy to be used as a cryptographic key.

8 PRIVACY-PRESERVING IN NEW GENERATIONS OF MOBILE CELLULAR NETWORKS

Our implementation is constructed in a manner that effectively prevents unauthorized access to information, providing a significant advantage in the construction of secure and dependable mobile telecommunication networks. To guarantee extensive network coverage, base stations are dispersed across diverse locations. A significant and inevitable vulnerability associated with this distribution is the inability to secure base stations against unauthorized access. Safeguarding sensitive user data is paramount for both the mobile network operator and individual users. Base stations are responsible for processing, caching, and storing various types of sensitive end-user data to provide low-latency performance or services to consumers. The physical location of the cached data is generally irrelevant to the end user, as long as the mobile service operates smoothly. The implementation of approaches such as Open Radio Access Network (RAN) further complicates the determination of data locality, as components of the Distributed Unit of the RAN can be virtualized at central or cell sites. Thus, it is increasingly important to prioritize the protection of private user data. Our FPGA-based architecture, incorporating PUFs, solves these issues while also enhancing the overall site-induced physical security risks. The existing base stations can easily integrate our architecture since most of them already use FPGA technologies. The hardware-level encryption provided by our approach is highly resistant to attacks, making it extremely difficult for attackers to bypass the security measures. Even if an attacker gains physical access to the base station, the encryption keys are required to access the data encrypted in memory.

9 CONCLUSION

In this work, we present a novel FPGA-based data-binding architecture using memory-based PUFs. Our architecture protects the confidentiality of data stored on NVM connected to the FPGA, and additionally binds the data to the memory module. Our design adopts a modular approach that enables customization of the overall implementation. It supports different memory modules and devices with different hardware resources in various use cases. Furthermore, we introduce a novel technique to increase the resistance against hardware attacks by generating random addresses used for PUF-readout. The concept is presented in a proof-of-concept implementation, demonstrated on five different devices, showing excellent performance in terms of low latency and area consumption. This implementation was tested on five different devices with varying performance capabilities and the PL/PS interconnectivity

on three different CPUs. The evaluation further contains the evaluation of three different cryptographic ciphers implemented as unrolled and round-based versions. To further improve our implementation and to take advantage of the AXI-streaming interface, the architecture could be extended by a pipelining approach to allow continuous reading and writing, which further reduces the latency, but is not considered in this work due to the cost of significantly higher area consumption. One further possibility for an outlook on potential improvements is the reduction of Trusted Computing Base (TCB) by masking mechanisms, as summarized in [40]. Finally, further research is needed to investigate challenges related to the robustness of PUFs, such as those arising from aging effects or potential failures in the memory module hosting the PUF.

ACKNOWLEDGMENTS

This work has been partially funded by the German Research Foundation – Deutsche Forschungsgemeinschaft (DFG), as part of the Project "PUFMem: Intrinsic Physical Unclonable Functions from Emerging Non-Volatile Memories" (project number 440182124). This work has additionally been accomplished within the projects "AUTOtech.agil" (FKZ01IS22088X) and "6G-RIC" (16KISK034). We acknowledge the financial support for the projects by the Federal Ministry of Education and Research of Germany (BMBF). At Virginia Tech, this project is partially supported by Commonwealth Cybersecurity Initiative, by the National Science Foundation (NSF) under grant 2153748, and by the Air Force Office of Scientific Research under award number FA9550-22-1-0548. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force. This work has been partially supported by the AMD Xilinx University Program (XUP). The authors would like to thank Aaron Difilippo and Calvin Hong from Virginia Tech for their contributions to three hardware-implemented cryptographic ciphers utilized in this manuscript.

REFERENCES

[1] Inc. Advanced Micro Devices. 2021. System Integrated Logic Analyzer v1.1. Retrieved April 15, 2024 from https://docs.xilinx.com/v/u/en-US/pg261-system-ila

[2] Inc. Advanced Micro Devices. 2023. 7 Series Product Tables and Product Selection Guide (XMP101). Retrieved April 15, 2024 from https://docs.xilinx.com/v/u/en-US/7-series-product-selection-guide

[3] Inc. Advanced Micro Devices. 2023. AXI DMA v7.1 LogiCORE IP Product Guide. Retrieved April 15, 2024 from https://docs.xilinx.com/r/en-US/pg021_axi_dma

[4] Nikolaos Athanasios Anagnostopoulos, Yufan Fan, Muhammad Umair Saleem, Nico Mexis, Emiliia Gelóczi, Felix Klement, Florian Frank, André Schaller, Tolga Arul, and Stefan Katzenbeisser. 2022. Testing Physical Unclonable Functions Implemented on Commercial Off-the-Shelf NAND Flash Memories Using Programming Disturbances. In *2022 IEEE 12th International Conference on Consumer Electronics (ICCE-Berlin)*. IEEE, IEEE, 445 Hoes Lane, Piscataway, NJ 08854, USA, 1–9. https://doi.org/10.1109/ICCE-Berlin56473.2022.10021310

[5] Jason H. Anderson. 2010. A PUF design for secure FPGA-based embedded systems. In *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, IEEE, 445 Hoes Lane, Piscataway, NJ 08854, USA, 1–6. https://doi.org/10.1109/ASPDAC.2010.5419927

[6] ARM. 2023. AMBA® AXI-Stream. Retrieved April 15, 2024 from http://aiweb.techfak.uni-bielefeld.de/content/bworld-robot-control-software/

[7] ARM. 2023. LogiCORE IP Clocking Wizard v3.2. Retrieved April 15, 2024 from https://docs.xilinx.com/v/u/en-US/clk_wiz_ds709

[8] Armin Babaei, Gregor Schiele, and Michael Zohner. 2022. Reconfigurable Security Architecture (RESA) Based on PUF for FPGA-Based IoT Devices. *Sensors* 22, 15 (2022). https://doi.org/10.3390/s22155577

[9] Alexandra Balan, Titus Balan, Marcian Cirstea, and Florin Sandu. 2020. A PUF-based cryptographic security solution for IoT systems on chip. *EURASIP Journal on Wireless Communications and Networking* 2020, 1 (Nov. 2020), 231. https://doi.org/10.1186/s13638-020-01839-6

[10] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. 2013. The SIMON and SPECK Families of Lightweight Block Ciphers. Cryptology ePrint Archive, Paper 2013/404.

[11] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. 2017. Notes on the design and analysis of SIMON and SPECK. *IACR Cryptol. ePrint Arch.* (2017), 560. http://eprint.iacr.org/2017/560

[12] Christophe Bobda, Joel Mandebi Mbongue, Paul Chow, Mohammad Ewais, Naif Tarafdar, Juan Camilo Vega, Ken Eguro, Dirk Koch, Suranga Handagala, Miriam Leeser, Martin Herbordt, Hafsah Shahzad, Peter Hofste, Burkhard Ringlein, Jakub Szefer, Ahmed Sanaullah, and Russell Tessier. 2022. The Future of FPGA Acceleration in Datacenters and the Cloud. *ACM Trans. Reconfigurable Technol. Syst.* 15, 3, Article 34 (2022), 42 pages. https://doi.org/10.1145/3506713

[13] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçın. 2012. PRINCE – A Low-Latency Block Cipher for Pervasive Computing Applications. In *Advances in Cryptology – ASIACRYPT 2012*, Xiaoyun Wang and Kazue Sako (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 208–225. https://doi.org/10.1007/978-3-642-34961-4_14

[14] Christoph Bösch, Jorge Guajardo, Ahmad-Reza Sadeghi, Jamshid Shokrollahi, and Pim Tuyls. 2008. Efficient Helper Data Key Extractor on FPGAs. In *Cryptographic Hardware and Embedded Systems – CHES 2008*, Elisabeth Oswald and Pankaj Rohatgi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 181–197. https://doi.org/10.1007/978-3-540-85053-3_12

[15] Dušan Božilov, Maria Eichlseder, Miroslav Knežević, Baptiste Lambin, Gregor Leander, Thorben Moos, Ventzislav Nikov, Shahram Rasoolzadeh, Yosuke Todo, and Friedrich Wiemer. 2021. PRINCEv2. In *Selected Areas in Cryptography (Lecture Notes in Computer Science)*, Orr Dunkelman, Jr. Jacobson, Michael J., and Colin O'Flynn (Eds.). Springer International Publishing, Cham, 483–511. https://doi.org/10.1007/978-3-030-81652-0_19

[16] Bertrand Cambou and Ying-Chen Chen. 2021. Tamper Sensitive Ternary ReRAM-Based PUFs. In *Intelligent Computing*, Kohei Arai (Ed.). Springer International Publishing, Cham, 1020–1040. https://doi.org/10.1007/978-3-030-80129-8_67

[17] David Castells-Rufas, Vinh Ngo, Juan Borrego-Carazo, Marc Codina, Carles Sanchez, Debora Gil, and Jordi Carrabina. 2022. A Survey of FPGA-Based Vision Systems for Autonomous Cars. *IEEE Access* 10 (2022), 132525–132563. https://doi.org/10.1109/ACCESS.2022.3230282

[18] Hao Chen, Yu Chen, and Douglas H. Summerville. 2011. A Survey on the Application of FPGAs for Network Infrastructure Security. *IEEE Communications Surveys & Tutorials* 13, 4 (2011), 541–561. https://doi.org/10.1109/SURV.2011.072210.00075

[19] Patrick Derbez and Léo Perrin. 2015. Meet-in-the-Middle Attacks and Structural Analysis of Round-Reduced PRINCE. In *Fast Software Encryption*, Gregor Leander (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 190–216. https://doi.org/10.1007/978-3-662-48116-5_10

[20] M. E. S. Elrabaa, M. Al-Asli, and M. Abu-Amara. 2021. Secure Computing Enclaves Using FPGAs. *IEEE Transactions on Dependable and Secure Computing* 18, 2 (2021), 593–604. https://doi.org/10.1109/TDSC.2019.2933214

[21] Michał Fularz, Dominik Pieczyński, and Marek Kraft. 2017. The performance comparison of the DMA subsystem of the Zynq SoC in bare metal and Linux applications. *Measurement Automation Monitoring* 63, 5 (2017), 189–191.

[22] Sezer Gören, Özgür Özkurt, Abdullah Yıldız, and H. Fatih Uğurdağ. 2011. FPGA bitstream protection with PUFs, obfuscation, and multi-boot. In *6th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE, IEEE, 445 Hoes Lane, Piscataway, NJ 08854, USA, 1–2. https://doi.org/10.1109/ReCoSoC.2011.5981541

[23] Matthias Hiller, Ludwig Kürzinger, and Georg Sigl. 2020. Review of error correction for PUFs and evaluation on state-of-the-art FPGAs. *Journal of Cryptographic Engineering* 10, 3 (Sept. 2020), 229–247. https://doi.org/10.1007/s13389-020-00223-w

[24] Brian Jarvis and Kris Gaj. 2017. Selection of an error-correcting code for FPGA-based physical unclonable functions. In *2017 International Conference on Field Programmable Technology (ICFPT)*. IEEE, IEEE, IEEE, 445 Hoes Lane, Piscataway, NJ 08854, USA, 243–246. https://doi.org/10.1109/FPT.2017.8280151

[25] Mohammad Nasim Imtiaz Khan, Chak Yuen Cheng, Sung Hao Lin, Abdullah Ash-Saki, and Swaroop Ghosh. 2021. A Morphable Physically Unclonable Function and True Random Number Generator Using a Commercial Magnetic Memory. *Journal of Low Power Electronics and Applications* 11, 1 (March 2021), 5. https://doi.org/10.3390/jlpea11010005

[26] Somayeh Kianpisheh and Tarik Taleb. 2023. A Survey on In-Network Computing: Programmable Data Plane and Technology Specific Applications. *IEEE Communications Surveys & Tutorials* 25, 1 (2023), 701–761. https://doi.org/10.1109/COMST.2022.3213237

[27] Stephan Kleber, Florian Unterstein, Matthias Hiller, Frank Slomka, Matthias Matousek, Frank Kargl, and Christoph Bösch. 2018. Secure Code Execution: A Generic PUF-Driven System Architecture. In *Information Security (Lecture Notes in Computer Science)*, Liqun Chen, Mark Manulis, and Steve Schneider (Eds.).

Springer International Publishing, Cham, 25–46. https://doi.org/10.1007/978-3-319-99136-8_2

[28] Miriam Leeser, Suranga Handagala, and Michael Zink. 2021. FPGAs in the Cloud. *Computing in Science & Engineering* 23, 6 (2021), 72–76. https://doi.org/10.1109/MCSE.2021.3127288

[29] Abhranil Maiti, Raghunandan Nagesh, Anand Reddy, and Patrick Schaumont. 2009. Physical unclonable function and true random number generator: a compact and scalable implementation. In *Proceedings of the 19th ACM Great Lakes Symposium on VLSI* (Boston Area, MA, USA) *(GLSVLSI '09)*. Association for Computing Machinery, New York, NY, USA, 425–428. https://doi.org/10.1145/1531542.1531639

[30] Priyanka Mall, Ruhul Amin, Ashok Kumar Das, Mark T. Leung, and Kim-Kwang Raymond Choo. 2022. PUF-Based Authentication and Key Agreement Protocols for IoT, WSNs, and Smart Grids: A Comprehensive Survey. *IEEE Internet of Things Journal* 9, 11 (2022), 8205–8228. https://doi.org/10.1109/JIOT.2022.3142084

[31] Nathan Menhorn. 2018. External secure storage using the PUF. Retrieved April 15, 2024 from https://docs.amd.com/r/en-US/xapp1333-external-storage-puf

[32] Kazuhiko Minematsu. 2007. Improved Security Analysis of XEX and LRW Modes. In *Selected Areas in Cryptography*, Eli Biham and Amr M. Youssef (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 96–113. https://doi.org/10.1007/978-3-540-74462-7_8

[33] Paul Newson. 2018. The application of FPGAs for wireless base-station connectivity. https://docs.amd.com/v/u/en-US/wp450-base-stn-connect

[34] Raluca Posteuca, Cristina-Loredana Duta, and Gabriel Negara. 2015. New approaches for round-reduced PRINCE cipher cryptanalysis. *Proceedings of the Romanian Academy, Series A* 16 (2015), 253–264.

[35] Pravin Prabhu, Ameen Akel, Laura M. Grupp, Wing-Kei S. Yu, G. Edward Suh, Edwin Kan, and Steven Swanson. 2011. Extracting Device Fingerprints from Flash Memory by Exploiting Physical Variations. In *Trust and Trustworthy Computing*, Jonathan M. McCune, Boris Balacheff, Adrian Perrig, Ahmad-Reza Sadeghi, Angela Sasse, and Yolanta Beres (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 188–201. https://doi.org/10.1007/978-3-642-21599-5_14

[36] Jerome Rampon, Renaud Perillat, Lionel Torres, Pascal Benoit, Giorgio Di Natale, and Mario Barbareschi. 2015. Digital Right Management for IP Protection. In *2015 IEEE Computer Society Annual Symposium on VLSI*. IEEE, IEEE, 445 Hoes Lane, Piscataway, NJ 08854, USA, 200–203. https://doi.org/10.1109/ISVLSI.2015.127

[37] Phillip Rogaway. 2004. Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC. In *Advances in Cryptology - ASIACRYPT 2004*, Pil Joong Lee (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 16–31. https://doi.org/10.1007/978-3-540-30539-2_2

[38] Ulrich Rührmair and Jan Sölter. 2014. PUF modeling attacks: An introduction and overview. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, IEEE, 445 Hoes Lane, Piscataway, NJ 08854, USA, 1–6. https://doi.org/10.7873/DATE.2014.361

[39] Sadman Sakib, Aleksandar Milenković, Md Tauhidur Rahman, and Biswajit Ray. 2020. An Aging-Resistant NAND Flash Memory Physical Unclonable Function. *IEEE Transactions on Electron Devices* 67, 3 (2020), 937–943. https://doi.org/10.1109/TED.2020.2968272

[40] Martin Schmid and Elif Bilge Kavun. 2023. Analyzing ModuloNET Against Transition Effects. In *2023 IEEE International Conference on Omni-layer Intelligent Systems (COINS)*. IEEE, IEEE, 445 Hoes Lane, Piscataway, NJ 08854, USA, 1–6. https://doi.org/10.1109/COINS57856.2023.10189305

[41] Johanna Sepulveda, Felix Willgerodt, and Michael Pehl. 2018. SEPUFSoC: Using PUFs for Memory Integrity and Authentication in Multi-Processors System-on-Chip. In *Proceedings of the 2018 on Great Lakes Symposium on VLSI* (Chicago, IL, USA) *(GLSVLSI '18)*. Association for Computing Machinery, New York, NY, USA, 39–44. https://doi.org/10.1145/3194554.3194562

[42] Joachim Strömbergson. 2023. aes. Retrieved April 15, 2024 from https://github.com/secworks/aes

[43] Manan Suri and Supriya Chakraborty. 2018. High-Quality PUF Extraction from Commercial RRAM Using Switching-Time Variability. In *2018 IEEE International Memory Workshop (IMW)*. IEEE, IEEE, 445 Hoes Lane, Piscataway, NJ 08854, USA, 1–4. https://doi.org/10.1109/IMW.2018.8388836

[44] Yinglei Wang, Wing-kei Yu, Shuo Wu, Greg Malysa, G. Edward Suh, and Edwin C. Kan. 2012. Flash Memory for Ubiquitous Hardware Security Functions: True Random Number Generation and Device Fingerprints. In *2012 IEEE Symposium on Security and Privacy*. IEEE, IEEE, 445 Hoes Lane, Piscataway, NJ 08854, USA, 33–47. https://doi.org/10.1109/SP.2012.12

[45] Florian Wilde, Berndt M. Gammel, and Michael Pehl. 2018. Spatial Correlation Analysis on Physical Unclonable Functions. *IEEE Transactions on Information Forensics and Security* 13, 6 (2018), 1468–1480. https://doi.org/10.1109/TIFS.2018.2791341

[46] Ke Xia, Yukui Luo, Xiaolin Xu, and Sheng Wei. 2021. SGX-FPGA: Trusted Execution Environment for CPU-FPGA Heterogeneous Architecture. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, IEEE, 445 Hoes Lane, Piscataway, NJ 08854, USA, 301–306. https://doi.org/10.1109/DAC18074.2021.9586207

[47] Wenjie Xiong, André Schaller, Nikolaos Athanasios Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Stefan Katzenbeisser, and Jakub Szefer. 2021. DRAM PUFs in Commodity Devices. *IEEE Design & Test* 38, 3 (2021), 76–83. https://doi.org/10.1109/MDAT.2021.3063370

[48] Shaza Zeitouni, Jo Vliegen, Tommaso Frassetto, Dirk Koch, Ahmad-Reza Sadeghi, and Nele Mentens. 2021. Trusted Configuration in Cloud FPGAs. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, IEEE, 445 Hoes Lane, Piscataway, NJ 08854, USA, 233–241. https://doi.org/10.1109/FCCM51124.2021.00036

[49] Jiliang Zhang, Yaping Lin, Yongqiang Lyu, and Gang Qu. 2015. A PUF-FSM Binding Scheme for FPGA IP Protection and Pay-Per-Device Licensing. *IEEE Transactions on Information Forensics and Security* 10, 6 (2015), 1137–1150. https://doi.org/10.1109/TIFS.2015.2400413

[50] Ji-Liang Zhang, Wei-Zheng Wang, Xing-Wei Wang, and Zhi-Hua Xia. 2017. Enhancing security of FPGA-based embedded systems with combinational logic binding. *Journal of Computer Science and Technology* 32 (2017), 329–339. https://doi.org/10.1007/s11390-017-1700-8

[51] Le Zhang, Zhi Hui Kong, Chip-Hong Chang, Alessandro Cabrini, and Guido Torelli. 2014. Exploiting Process Variations and Programming Sensitivity of Phase Change Memory for Reconfigurable Physical Unclonable Functions. *IEEE Transactions on Information Forensics and Security* 9, 6 (2014), 921–932. https://doi.org/10.1109/TIFS.2014.2315743

[52] Xian Zhang, Guangyu Sun, Yaojun Zhang, Yiran Chen, Hai Li, Wujie Wen, and Jia Di. 2016. A novel PUF based on cell error rate distribution of STT-RAM. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, IEEE, 445 Hoes Lane, Piscataway, NJ 08854, USA, 342–347. https://doi.org/10.1109/ASPDAC.2016.7428035

[53] Mark Zhao, Mingyu Gao, and Christos Kozyrakis. 2022. ShEF: Shielded Enclaves for Cloud FPGAs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 1070–1085. https://doi.org/10.1145/3503222.3507733

[54] Jason X. Zheng, Dongfang Li, and Miodrag Potkonjak. 2014. A secure and unclonable embedded system using instruction-level PUF authentication. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, IEEE, IEEE, 445 Hoes Lane, Piscataway, NJ 08854, USA, 1–4. https://doi.org/10.1109/FPL.2014.6927428