

PUF ROKs : A Hardware Approach to Read-Once Keys

Michael S. Kirkpatrick
Department of Computer
Science
Purdue University
West Lafayette, IN 47907
mkirkpat@cs.purdue.edu

Sam Kerr
Department of Computer
Science
Purdue University
West Lafayette, IN 47907
stkerr@cs.purdue.edu

Elisa Bertino
Department of Computer
Science
Purdue University
West Lafayette, IN 47907
bertino@cs.purdue.edu

ABSTRACT

Cryptographers have proposed the notion of read-once keys (ROKs) as a beneficial tool for a number of applications, such as delegation of authority. The premise of ROKs is that the key is destroyed by the process of reading it, thus preventing subsequent accesses. While the idea and the applications are well-understood, the consensus among cryptographers is that ROKs cannot be produced by algorithmic processes alone. Rather, a trusted hardware mechanism is needed to support the destruction of the key. In this work, we propose one such approach for using a hardware design to generate ROKs. Our approach is an application of physically unclonable functions (PUFs). PUFs use the intrinsic differences in hardware behavior to produce a random function that is unique to that hardware instance. Our design consists of incorporating the PUF in a feedback loop to make reading the key multiple times physically impossible.

Categories and Subject Descriptors

K.6.5 [MANAGEMENT OF COMPUTING AND INFORMATION SYSTEMS]: Security and Protection—*physical security*

General Terms

Security

Keywords

Physically Unclonable Functions, Applied Cryptography, Access Control, Read-Once Keys, Hardware Design

1. INTRODUCTION

The term read-once keys (ROKs) describes the abstract notion that a cryptographic key can be read and used for encryption and decryption only once. While it seems intuitive that a trusted piece of software could be designed that deletes a key right after using it, such a scheme naïvely

depends on the proper execution of the program. This approach could be easily circumvented by running the code within a debugging environment that halts execution of the code before the deletion occurs. That is, the notion of a ROK entails a stronger protection method wherein the process of reading the key results in its immediate destruction.

ROKs could be applied in a number of interesting scenarios. One application could be to create one-time programs [18], which could be beneficial for protecting the intellectual property of a piece of software. A potential client could download a fully functional one-time program for evaluation before committing to a purchase. A similar application would be self-destructing email. In that case, the sender could encrypt a message with a ROK; the message would then be destroyed immediately after the recipient reads the message. More generally, there is considerable interest in self-destructing data, both commercially [3] and academically [17]. In addition, the use of trusted hardware tokens have been proposed for applications including program obfuscation [29], monotonic counters [33], oblivious transfer [27], and generalized secure computation [19]. ROKs can provide the required functionality for these applications.

Another interesting application of PUF ROKs is to defend against physical attacks on cryptographic protocols. For example, consider fault injection attacks on RSA [11, 10, 9, 8, 30]. In these attacks, the algorithm is repeatedly executed with the same key, using a controlled fault injection technique that will yield detectable differences in the output. After enough such iterations, the attacker is able to recover the key in full. Similarly, “freezing” is another class of physical attack that can extract a key if it was *ever* stored in an accessible part of memory [5]. PUF ROKs offer a unique defense against all of these attacks because repeated execution with the same key cannot occur, and the key is *never* actually present in addressable physical memory.

The ability to generate ROKs in a controlled manner could also lead to an extension where keys can be generated and used a multiple, but limited, number of times. For example, consider the use of ROKs to encrypt a public key pk . If an identical ROK can be generated twice, the owner of pk could first use the key to create $e_{ROK}(pk)$ (indicating the encryption of pk under with the ROK). Later, an authorized party could create the ROK a second time to decrypt the key. Such a scheme could be used to delegate the authority to cryptographically sign documents.

In a sense, a ROK is an example of a program obfuscator. An obfuscator \mathcal{O} takes a program \mathcal{P} as input and returns $\mathcal{O}(\mathcal{P})$, which is functionally identical to \mathcal{P} but inde-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASIACCS '11, March 22–24, 2011, Hong Kong, China.

Copyright 2011 ACM 978-1-4503-0564-8/11/03 ...\$10.00.

cipherable. A ROK, then, involves an obfuscator that makes only the key indecipherable. While ROKs are promising ideals, the disheartening fact is that program obfuscators—of which ROKs are one example—cannot be created through algorithmic processes alone [7]. Instead, trusted hardware is required to guarantee the immediate destruction of the key [18]. However, we are aware of no work that has specifically undertaken the task of designing and creating such trusted hardware for the purpose of generating a ROK.

In this paper, we propose the creation of ROKs using physically unclonable functions (PUFs) [15, 16]. A PUF takes an input *challenge* $C_i \in C$, where C denotes the set of all such possible challenges. The PUF then produces a *response* $R_i \in R$, where R is the set of possible responses. The function that maps each C_i to R_i is based on the intrinsic randomness that exists in hardware and *cannot be controlled*. As such, an ideal PUF creates a mathematical function unique to each physical instance of a hardware design; even if the same design is used for two devices, it is physically impossible to make their PUFs behave identically.

Our insight for the design of such “PUF ROKs” is to incorporate the PUF in a feedback loop for a system-on-chip (SoC) design.¹ That is, our design is for the PUF to reside on the same chip as the processor core that performs the encryption. This integration of the PUF and the processor core protects the secrecy of the key. An attempt to read the key from memory (given physical access) will fail, because the key *never exists in addressable memory*. Also, attempts to learn the key from bus communication will be difficult or impossible, as each key is used to encrypt only a single message, and the key is *never transmitted across the bus*.

The unpredictable nature of PUFs provides a high probability that each iteration of a ROK generation will produce a unique, seemingly random key. Yet, to ensure that a key can be generated to perform both encryption and decryption, the PUF must be initialized repeatedly to some state, thus providing the same sequence of keys. To accomplish this, Alice could provide an initial seed to produce a sequence of keys that are used to encrypt a set of secrets. Alice could then reset the seed value before making the device available to Bob. Bob, then, could use the PUF to recreate the keys in order, decrypting the secrets. As Bob has no knowledge of the seed value, he is unable to reset the device and cannot recreate the key just used.

Astute readers will note the similarities between our approach and using a chain of cryptographic hashes to generate keys. That is, given a seed x_0 , the keys would be $H(x_0)$, $H(H(x_0))$, etc., where H denotes a cryptographic hash function. The insight of our approach is that a PUF, as a trusted piece of hardware, can provide a hardware-based implementation that is analogous to a hash function, but is more secure than software implementations of such algorithms.

The rest of this paper is organized as follows. Section 2 discusses PUFs and related works that employ them. Section 3 formalizes the notion of a ROK. We continue this formalization in Section 4, in which we prove that our design captures the essence of a ROK and describe extensions of our approach for greater flexibility. We present the details of our implementation in Section 5, and offer a security analysis in Section 6, before concluding in Section 7.

¹Our design could also be made to work for application-specific integrated circuits (ASICs), but we limit our discussion to SoC designs for simplicity.

2. PUFs

Before we describe our design for PUF ROKs, we provide some relevant background on the creation, properties, and applications of PUFs. Research on PUFs [15, 16] arose from the observation that distinct instances of hardware produce unique behavioral characteristics [28]. That is, each copy of the device, even if designed to be identical, will exhibit slight variations that can be measured only during execution of the circuit. The precise behavior that ensues can be neither controlled nor predicted. In addition to silicon-based circuits, similar distinguishing techniques have been applied to RFID devices [13, 12].

Mathematically, a PUF can be modeled as a function $PUF : C \rightarrow R$, where C denotes a set of input challenges (usually encoded as a bit string) and R is a set of responses. That is, for a single device, providing the same input C_i to the PUF will yield approximately the same result R_i . In practice, the PUF output consists of noisy data; error-correcting codes, such as Reed-Solomon [31], can be applied to ensure that the response is identical every time.

The definition of the function is determined exclusively by the variations in the hardware. As a result, no properties can be assumed about the function itself. That is, in general, PUFs are neither linear nor injective nor surjective. Rather, the function merely consists of a set of random pairs (C_i, R_i) . Furthermore, as the function is defined by the hardware, providing the same C_i as input to a different device’s PUF will produce a different response $R'_i \neq R_i$.

The unpredictable nature of the PUF behavior leads to a complication for proofs of security. Specifically, it is possible that the function approaches a degenerate case. For example, it is possible that the challenge input C_i will produce itself as the response. One way to counteract this problem is to incorporate a hash function into the structure; that is, the output of the PUF is immediately hashed, and the output of the hash becomes R_i . To simplify our proofs later, we will assume an **ideal PUF**, meaning that there is no predictable connection between the challenges and responses. That is, assume an observer has knowledge of (C_i, R_i) and another challenge C'_i , where C_i and C'_i differ in only a single bit, the observer can predict the corresponding R'_i with only negligible probability. Furthermore, the ideal PUF assumption means that $C_i = R_i$ for any i with only negligible property.

As an example of a circuit-based PUF, consider the design in Figure 1. In this design, the one-bit challenge input controls the multiplexor (MUX) and redirects the output of an oscillator to a different counter. For instance, if $C_i = 1$, the top oscillator will be directed to the top counter, while the bottom counter captures the bottom oscillator’s frequency. If $C_i = 0$, then the top counter captures the bottom oscillator, and vice versa. The oscillators are allowed to run for a certain duration (using the same C_i for the entire time), after which the counters’ values are compared. If the top counter reports a larger value, the response $R_i = 1$. Otherwise, $R_i = 0$. While this design produces only a single bit of output, larger PUFs can generate longer bit strings.

In this PUF, the ring oscillators are designed to be identical. However, as a result of the manufacturing process, the wire length and width will inevitably differ. Hence, one oscillator will switch between outputting a 1 and a 0 at a faster rate. But without actually executing the circuit, it is impossible to tell which oscillates faster. This is due to the fact that the difference between the two oscillators is too

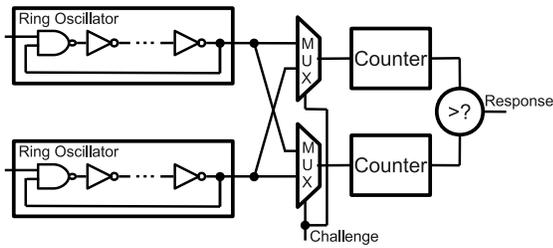


Figure 1: A sample 1-bit ring oscillator PUF

small to be measured. Thus, one cannot control or predict the output of this PUF just by inspecting the device.

PUFs have been applied in a number of settings. One common approach is to use the response to provide secure cryptographic key storage [21, 20]. For instance, to protect the key \mathcal{K} , one could compute $X = \mathcal{K} \oplus R_i$, where \oplus denotes the bitwise XOR operator. As R_i is a random bit string, it acts as a one-time pad and the value X can then be stored in plaintext without sacrificing the confidentiality of \mathcal{K} .

Other applications have also been proposed. One technique uses PUFs to bind software to hardware in a VM environment [6]. The AEGIS secure processor [35, 36] incorporates a PUF for key generation and storage. Robust PUFs have been proposed as an authentication scheme for banking environments [14]. Finally, in previous work, we have proposed the use of PUFs to combat insider threats by binding authentication to trusted devices [22, 23]. In addition, we have presented preliminary design ideas underlying this work as an extended abstract [24]; however, the current paper significantly expands on this work, adding a prototype implementation, security analysis, formal definitions, and extensions for out-of-order processing of PUF ROKs.

3. ROKS

Our formal notion of a ROK is based on an adaptation of Turing machines. Specifically, define the machine T to be

$$T = \langle Q, q_0, \delta, \Gamma, \iota \rangle$$

where Q is the set of possible states, q_0 is the initial state, δ defines the transition from one state to another based on processing the symbols Γ , given input ι . Readers familiar with Turing machines will note that ι is new. In essence, we are dividing the traditional input symbols into code (Γ) and data (ι). For the sake of simplicity, we assume that ι only consists of messages to be encrypted or decrypted and ignore other types of input data. Thus, the definition of δ is determined by the execution of instructions $\gamma_1, \gamma_2, \dots, \gamma_i$, where consuming $\gamma_i \in \Gamma$ results in the transition from state q_i to q_{i+1} . Based on this formalism, we propose the following primitives.

- The **encrypt primitive** $\text{Enc}(\gamma_i, q_i, m)$ encrypts the message $m \in \iota$ given the instruction γ_i and the state q_i . The system then transitions to q_{i+1} and produces the returned value as $e(m)$ as a side effect.
- The **decrypt primitive** $\text{Dec}(\gamma_j, q_j, e)$ decrypts the ciphertext $e \in \iota$ given the instruction γ_j and the state q_j . If the decryption is successful, the primitive returns m . Otherwise, the return value is denoted \emptyset . The system then transitions to q_{j+1} .

Informally, γ_i and q_i describe the current instruction and the contents of memory for a single execution of a program, and capture the state of the system just before executing the encrypt or decrypt primitive. That is, if the execution of the program is suspended for a brief time, γ_i, q_i would describe a snapshot of the stack, the value stored in the instruction pointer (IP) register, the values of all dynamically allocated variables (*i.e.*, those on the heap), etc. In short, it would contain the full software image for that process for that precise moment in time. Once the program is resumed, the symbol γ_i would be consumed, and the system would transition to state q_{i+1} . Given these primitives, we present the following definition.

Definition: A **read-once key** (ROK) is a cryptographic key \mathcal{K} subject to the following conditions:

- Each execution of $\text{Enc}(\gamma_i, q_i, m)$ generates a new \mathcal{K} and yields a transition to a unique q_{i+1} .
- The first execution of $\text{Dec}(\gamma_j, q_j, e)$ returns m and transitions to q_{j+1} . All subsequent executions return \emptyset and transitions to q'_{j+1} , even when executing the machine $\langle Q, q_0, \delta, \Gamma, \iota \rangle$ with e , except with negligible probability.
- The probability of successfully decrypting e without the primitive $\text{Dec}(\gamma_j, q_j, e)$ is less than or equal to a security parameter ϵ ($0 < \epsilon < 1$), even when given identical initial states. ϵ must be no smaller than the probability of a successful attack on the cryptographic algorithms themselves.

What these definitions say is that the ROK Turing machine is non-deterministic. Specifically, during the first execution of a program² that encrypts a message m , δ will define a transition from q_i to q_{i+1} based on the primitive $\text{Enc}(\gamma_i, q_i, m)$. However, the second time, the key will be different, and the state transition will be from q_i to q'_{i+1} . Similarly, the first execution of a program that decrypts $e(m)$ will traverse the states q_0, \dots, q_j, q_{j+1} , where q_{j+1} is the state that results from a successful decryption. However, returning the machine to its initial state q_0 , using the same instructions Γ , the state traversal will be $q_0, \dots, q_j, q'_{j+1} \neq q_{j+1}$, because the decryption fails. Thus, ROKs incorporate some unpredictable element that does not exist in traditional Turing machines: the history of prior machine executions. That is, for any given machine T , only the first execution (assuming either the encrypt or decrypt primitive is executed) will use the transitions defined by δ . The second (and subsequent) executions will use δ' , as the state after the primitive is invoked will differ.

Clearly, these definitions capture the intuitive notion of a ROK. The key \mathcal{K} is generated in an on-demand fashion in order to encrypt a message. Later, \mathcal{K} can be used to decrypt the message, but only once. After the first decryption, the key is obliterated in some manner. Specifically, even if the contents of memory are returned to match the program state γ_j, q_j as it existed before the first call to

²Observe that the program doing the encryption is separate from the one doing the decryption. If the encryption and decryption occurred in the same program, the decryption would succeed, as the key would have just been dynamically generated. In contrast, when the programs are distinct, only the first execution of the decryption program will succeed.

$\text{Dec}(\gamma_j, q_j, e)$, the decryption will fail. The intuition here is that a special-purpose hardware structure must provide this self-destructing property.

Observe that an adversary \mathcal{A} may opt to attack the cryptographic algorithms themselves. In such an attack, the number of times the key \mathcal{K} can be read by an authorized party is irrelevant: \mathcal{A} is never authorized. If the cryptographic scheme is sufficiently weak, \mathcal{A} may succeed in recovering the message (or the key itself). The ROK property offers no additional security against such an attack. That is, we are making no special claims of cryptographic prowess. For this reason, we require that ϵ be no smaller than the probability of a successful attack on the cryptographic scheme employed.

What is unique about our technique is that we are offering a means to limit the usage of a key by an authorized party. Clearly, with sufficient motivation, this authorized party may become an adversary himself, attempting to recover the key \mathcal{K} and subvert the system. The parameter ϵ offers a means to specify the system’s defense against such an insider threat. For the most sensitive data, an implementation of our design could require a very low level of ϵ , making the probability of subverting the ROK property equal to the probability of a brute-force attack on the cryptographic algorithm. In applications that are less sensitive (*i.e.*, the ROK property is desirable, but not critically important), ϵ could be larger. In short, ϵ captures the flexibility to adjust the security guarantees of the ROK according to desired implementation characteristics. We will explore this idea more in Sections 4 and 6.

4. PUF ROKS

In this section, we propose the use of PUFs to generate ROKs, which we call PUF ROKs. Like previous work [34], our design is based on the idea of using the PUF output to generate a transient key dynamically. We start this section by describing the basic hardware architecture for creating a PUF ROK component. We then proceed to prove formally that this architecture captures the desired ROK characteristics. This section concludes with descriptions of how PUF ROKs can be used in both symmetric key and public key cryptography.

4.1 PUF ROK Overview

The high-level view of our hardware architecture for generating PUF ROKs consists of a number of components. We formally define these components and their functional connections as follows.

- The **processor core** (PC) executes the desired application. The PC has access to volatile random access memory (RAM) for implementing the typical C-language execution constructs, such as a stack and heap. The PC contains an interface to a physically distinct **crypto core** (CC).
- The CC is a stand-alone hardware component that provides cryptographic services to the PC. The CC provides the following service interface to the PC:
 - $\text{Init}(x_0)$: an initialization routine that takes an input x_0 as a seed value for the PUF. There is no return value.

- $\text{Enc}(m)$: an encryption primitive that takes a message m as input and returns the encrypted value $e(m)$.
- $\text{Dec}(e(m))$: a decryption primitive that takes a ciphertext as input. Given $e(m)$ repeatedly, this service returns the plaintext m only on the first execution. Subsequent calls to this service for $e(m)$ return \emptyset .

- The CC has a unidirectional interface with a **register** (Reg). Whenever the CC’s $\text{Init}(x_0)$ service is invoked, the CC writes x_0 (or a value derived from x_0 , if so desired) into Reg.
- The CC can poll the **PUF**. When this occurs, the value stored in Reg is used as the PUF challenge. The response is then fed into an **error correction unit** (ECU). After performing mode-specific functions, the ECU returns a sanitized PUF output to the CC, while simultaneously overwriting the contents of Reg. When decrypting, the write back to Reg is contingent on feedback from CC. That is, Reg would only be overwritten during $\text{Dec}(e(m))$ if the decryption was successful.

Figure 2 shows a high-level view of a SoC implementation of our PUF ROK design. The key insight of this approach is the the PUF-ECU-Reg portion form a feedback loop. The PUF uses the values stored in the Reg as its input challenge C_i . The raw response R_i is run through an error correction routine to produce R'_i , which is written back into the Reg. The cleaned response is also reported to the CC for use in the cryptographic operations.

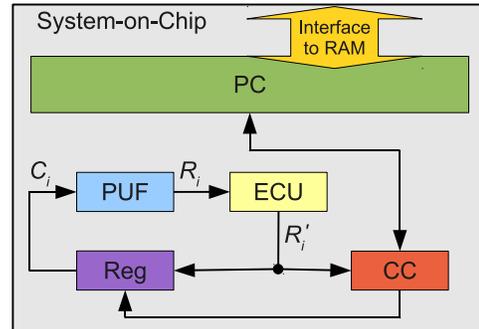


Figure 2: Components for a SoC PUF ROK design

The operation of the ECU depends on the cryptographic primitive invoked. In the case of encryption, the key \mathcal{K} is just being created. As such, there are no errors to correct. Instead, the ECU uses the raw PUF output as the “correct” value and generates a small amount of error-correcting data. This data is stored in a local cache that is accessible only to the ECU itself. Later, when decryption occurs, this data is used to correct any transient bit errors that may have occurred during the PUF execution. Observe that, as the error correction occurs *before* the response is stored in the Reg, this design structure ensures that the challenge inputs are correct.

The security parameter ϵ is used to specify the size of x_0 . Specifically, to meet the security guarantees dictated by the ROK formalism, x_0 must be at least $\lceil -\log_2 \epsilon \rceil$ bits³ in

³Recall that $0 < \epsilon < 1$. As a result, $\log_2 \epsilon < 0$, so $\lceil -\log_2 \epsilon \rceil$ is a nonnegative integer.

length, with each bit distributed uniformly. For completeness, we assume that x_0 has a length of at least one bit (for the cases when $\epsilon > 1/2$). Our subsequent analysis holds, but including this fact in later proofs adds unnecessary mess to the notation. As such, we will omit this fact and implicitly assume that x_0 is at least one bit in length.

In this architecture, we make two simplifying assumptions. First, we assume that the challenges and responses are the same length. We also assume that Reg consists of a small storage cache of the same size.⁴ In implementations where these assumptions do not hold, additional hardware components may be required.

4.2 System Details

Our architecture, as previously proposed, seems rather limited in use. Primarily, keys must be used for decryption in the same order as they are created. For example, consider the case where two messages m_1 and m_2 are encrypted. The first encryption would generate \mathcal{K}_1 and the second creates \mathcal{K}_2 . To switch to decrypt mode, the $\text{Init}(x_0)$ primitive would be required to return the Reg to its original state. The implication of this design is that the user must perform $\text{Dec}(e(m_1))$ before $\text{Dec}(e(m_2))$.

An intuitive solution would be to pass a counter n along with the $\text{Dec}(e(M))$ invocation, indicating that the PUF must be polled n times before the appropriate key would be reached. Hence, to decrypt the second message first, the invocation would be $\text{Dec}(e(m_2), 2)$. This solution is problematic, though. Specifically, once $\text{Dec}(e(m_1), 2)$ is invoked, the contents of Reg would no longer contain x_0 , and there would be no way for $\text{Dec}(e(m_1), 1)$ to generate \mathcal{K}_1 .

A similar problem is that any switch between encryption and decryption would require resetting the contents of Reg and polling the PUF multiple times. For instance, assume the user has encrypted three messages m_1, m_2 , and m_3 . The PUF ROK would have generated keys $\mathcal{K}_1, \mathcal{K}_2$, and \mathcal{K}_3 . To decrypt $e(m_1)$, $\text{Init}(x_0)$ restores the Reg to its initial state, and $\text{Dec}(e(m_1))$ is invoked. After the decryption, Reg is storing R_1 . In order to encrypt message m_4 , the PUF would need to be polled two more times to ensure that key \mathcal{K}_4 is generated. This can become very complicated to maintain.

To address these problems, we expand the details of our design as shown in Figure 3. In this figure, we partition the high-level Reg into distinct registers for processing the challenge input for encryption (EncReg) and for decryption (DecReg), as well as one that stores the seed value (SeedReg). We also introduce an error-correcting cache (EC Cache). The intuition in this design is that the ECU will store n error-correcting codes that can be accessed in arbitrary order. Once the first k codes have been used, they can be replaced. When this happens, the ECU synchronizes with the SeedReg (as indicated by the Sync line).

The operation of our PUF ROK architecture can be illustrated by the following example. Assume $n = 4$. A sample work flow could be as follows:

1. The user initializes the system with $\text{Init}(x_0)$.
2. Three messages, m_1, m_2 , and m_3 are encrypted. The keys $\mathcal{K}_1, \mathcal{K}_2$, and \mathcal{K}_3 are derived from the PUF re-

⁴Depending on the size of the PUF output, Reg may correspond to an array of hardware registers. *E.g.*, if the PUF output is 256 bits and the hardware registers store only 32 bits each, then Reg consists of eight physical registers.

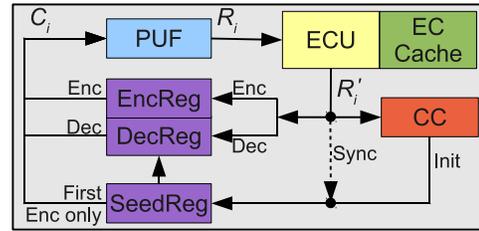


Figure 3: Extension components for out-of-order PUF ROKS

sponses R_1, R_2 , and R_3 , respectively. The ECU stores error correcting codes EC_1, EC_2 , and EC_3 in its cache.

3. Message m_2 is decrypted by invoking $\text{Dec}(e(m_2), 2)$. The contents of SeedReg are copied into DecReg, and the PUF is polled twice to generate R_2 and the corresponding \mathcal{K}_2 . The ECU marks EC_2 as invalid, assuming the decryption is successful.
4. Message m_4 is encrypted, using R_4 and \mathcal{K}_4 . EC_4 is generated and stored in the cache.
5. Message m_1 is decrypted by $\text{Dec}(e(m_1), 1)$. At this point, as both EC_1 and EC_2 have become invalid, the ECU initiates the Sync action.
6. During the Sync, the ECU takes control of the PUF feedback loop. The PUF is polled twice, using the contents of SeedReg as the challenge input (x_0). As a result, responses R_1 and R_2 are generated, and R_2 is ultimately stored in SeedReg. As a result of Sync, the part of EC Cache that was used to store EC_1 and EC_2 is now marked as available.
7. To encrypt m_5 , $\text{Enc}(m_5)$ is invoked like normal, using the contents of EncReg as the challenge input. The corresponding EC_5 is then stored in one of the newly available slots in EC Cache.

While this approach addresses the complication of using keys out of order, a simple extension makes the design even more powerful. Consider the case where a key is needed n times, rather than just once. *E.g.*, if Alice needs Bob to encrypt 10 messages on her behalf, she could either use 10 ROKs or she could employ, for lack of a better term, a read-10-times-key. The extension above, with the integrated EC Cache, could accommodate this request by storing the EC_i codes 10 times. The codes would then become invalid after all 10 slots in the EC Cache have been marked as such.

4.3 Formal Proof of PUF ROK

While it may seem intuitive that our architectural design captures the essence of a ROK, we present the following formal proof to illustrate the use of the security parameter ϵ . In this theorem, we use the notation $|s|$ to denote the length of the bit string s . To simplify the proof, we elide the details discussed in Section 4.2, and simply use the high-level architectural terms from Section 4.1.

Theorem: Assuming an ideal PUF and adequate error correction are employed, the PUF ROK architectural design successfully enforces the ROK criteria.

Proof: To demonstrate the claim, we must show that all three properties of ROKs hold. Clearly, our definitions of the $\text{Enc}(m)$ and $\text{Dec}(e(m))$ services provided by the CC match those of the ROK definition. However, to see that the first two ROK properties hold, we must consider the interaction of the components. When the $\text{Enc}(m)$ service is invoked, the contents of Reg are used as input to the PUF and replaced with the output. This output is also used to generate the key \mathcal{K} . Thus, the first property holds.

To switch the PUF ROK to decrypt mode, the seed x_0 must again be written to Reg. This action must be done by the encrypting party. After this is done, the $\text{Dec}(e(m))$ primitive will involve polling the PUF, thus using x_0 as the PUF’s input challenge once again. The ECU ensures that the new PUF result matches the previous output, and the decryption key \mathcal{K} will be identical to that used for encryption. However, the new PUF result also replaces x_0 in the Reg. Hence, as the PUF is assumed to be ideal, the first execution of $\text{Dec}(e(m))$ will produce the correct key \mathcal{K} . Any future execution of $\text{Dec}(e(m))$ will, with near certainty, produce an incorrect key; the CC will then return \emptyset . Thus, the ideal PUF assumption ensures that the second criterion holds.

To complete the proof, we must show that our design satisfies the third criterion, which is that the probability of bypassing the restriction on the $\text{Dec}(\gamma_j, q_j, e)$ primitive is less than or equal to ϵ . There are three ways this could occur. First, the adversary \mathcal{A} could succeed in an attack on the cryptographic scheme itself. However, by definition, the probability of this occurring is guaranteed to be no more than ϵ , and the third criterion would hold. The second possibility would be for the ECU to produce the key \mathcal{K} erroneously. However, this violates our assumption that adequate error correction is employed. Hence, this case cannot occur.

The third case is for a later execution of the PUF to produce x_0 as a result. That is, there must exist some j such that the contents of Reg would be the sequence

$$x_0, R_1, R_2, R_3, \dots, R_j = x_0$$

Recall that we assumed ($m \geq 1$)

$$|C_j| = |R_j| = |R'_j| = |x_0| = m \geq \lceil -\log_2 \epsilon \rceil$$

Then the probability that any $R_j = x_0$ would be $1/2^m$, which gives us

$$P(R_j = x_0) = \frac{1}{2^m} \leq \frac{1}{2^{\lceil -\log_2 \epsilon \rceil}} \leq \frac{1}{2^{\log_2 \epsilon^{-1}}} = \frac{1}{\epsilon^{-1}} = \epsilon$$

That is, the probability that any polling of the PUF will produce an output that matches the seed x_0 is less than the security parameter ϵ . Hence, the third required property of ROKs also holds. Thus, the PUF ROK architecture successfully implements the ROK requirements under the ideal PUF assumption. \square

4.4 Symmetric Key PUF ROKs

The functionality of the CC depends on the cryptographic application. In the preceding discussion, we were focusing on a symmetric key application, in essence. However, there are a few more details regarding the operation of the CC that we must address here.

As noted above, in symmetric key cryptographic applications, the PC issues the command $\text{Enc}(m)$, where m indicates the message to be encrypted. The CC then issues the command $\text{Init}(x_0)$, which writes the value x_0 into the Reg. The PUF then uses x_0 as C_1 and generates R_1 , which is passed to the ECU. The ECU then generates the necessary error-correcting codes EC_1 to ensure the key is later recoverable, even if the noisy output of the PUF produces a small number of bit errors.

Next, to guarantee a strong key from R_1 , the CC applies a cryptographic hash. That is, to generate a 256-bit AES key, the CC computes $\mathcal{K}_1 = H(R_1)$, where H is the SHA-256 hash function. While an ideal PUF is assumed to produce random mappings, we employ the hash function in this way to add to the entropy of the system. That is, if R_i and R_j (the responses produced by two different PUF pollings) differ by only a single bit, $H(R_i)$ and $H(R_j)$ will have a Hamming distance of 128 bits on average. As a result, even if an attacker is able to recover the key just by observing the plaintext and ciphertext, the hash prevents modeling the PUF, as doing so would require the attack to create a pre-image of the hash.

Once the CC has polled the PUF and generated the key, the encrypted message $e(m)$ is provided to the PC. Later, when the recipient wishes to decrypt the message (which can only be done once), the PC issues the command $\text{Dec}(e(m))$ to the CC. The CC then resets the Reg with $\text{Init}(x_0)$, and polls the PUF to recreate the key. The decrypted message, then, is returned to the PC.

For the sake of completeness, observe that we have never detailed how x_0 is determined. One approach, which depends only on the device itself, would be to take the timestamp ts when the PC invokes the Init primitive, and uses $x_0 = H(ts)$. In another approach, the PC could use a user’s password, and hash it similarly. Thus, the seed value can be determined in multiple ways.

4.5 Public Key PUF ROKs

Now consider the case of public key cryptography. In this setting, we start with the assumption that the CC contains the necessary parameters for the public key computations. For instance, if the RSA cryptosystem is used, the CC knows (or can create) the two large prime numbers p and q such that $n = pq$.⁵ The goal, then, is to generate a pair (pk, sk) , where pk denotes a public key and sk denotes the corresponding private key.

In contrast to the symmetric key approach, the CC does not need to generate the ROK twice. As such, the $\text{Init}(x_0)$ function is optional. However, the CC still polls the PUF to generate the pair of keys. The challenge with using a PUF to create a public key pair, though, is how to generate a bit string that is long enough. A strong RSA key, for example, is at least 2048 bits long. But creating a 2048-bit PUF output would require a prohibitively large circuit design.

Instead, our approach is for the CC to buffer a series of PUF results. For instance, if the PUF produces a 256-bit output, the CC could use R_i as bits 0-255, $R_i + 1$ as bits

⁵Readers familiar with the properties of RSA will observe that it is necessary for the CC to create and store the primes p and q securely. Such functionality is common in existing cryptographic processors, such as a TPM. Consequently, we assume that the CC is designed to ensure the secrecy of p and q is preserved, and omit further discussion of this matter.

255-511, and so forth. Once the CC has polled the PUF to get a sufficient number of random bits, the next challenge is to convert this bit string into a strong key. For simplicity, we assume the use of RSA.

Let e denote the candidate key that the CC has received from the PUF. In order to use e as an RSA key, e must be coprime to $\phi(n) = (p-1)(q-1)$. By applying the Euclidean algorithm, the CC can compute the greatest common divisor $g = \gcd(e, \phi(n))$. If $g = 1$, e and $\phi(n)$ are coprime, and e can be used as is. Otherwise, $e' = e/g$ can be used. The secret key sk , then becomes e or e' as appropriate. To compute the public key pk , the CC computes the modular multiplicative inverse of sk by using the extended Euclidean algorithm. That is, the CC computes d such that $sk \cdot d \equiv 1 \pmod{\phi(n)}$. This value d then becomes the public key pk . Given this key pair (pk, sk) , the PUF ROK can be used by the PC in multiple ways. First, the PC could issue the command `Sign(m)` to the CC, requesting a cryptographic signature. After generating (pk, sk) , the CC uses sk to sign m , returning the signature and the public key pk to PC. pk can then be used by a third party to verify the signature.

Alternatively, the PC could issue the command `Gen`, which tells the CC to generate the key pair. Instead of using the keys immediately, the CC stores sk and returns pk to the PC. A third party wishing to send an encrypted message to the PC could use pk as needed. Then, the PC would issue `Dec(m)` to have the CC decrypt the message. While this violates the spirit of the ROK principle (as sk would need stored somewhere), sk could simply be thrown away during the `Gen` procedure. Later, when the decryption occurs, the sk would be recreated, making the public key PUF ROK work similarly to the symmetric key version.

Finally, consider the case where the third party needs assurance that the public key pk did, in fact, originate from the PUF ROK. This can be accomplished if the CC contains a persistent public key pair, similar to the Endorsement Key (EK) stored in a Trusted Platform Module (TPM). In addition to providing the pk to the PC, the CC could also return `SignEK(pk)`, denoting the signature of the pk under this persistent key. This technique provides the necessary assurance, as the persistent key is bound to the CC. However, this requires a key management infrastructure similar to existing TPM-based attestation schemes.

4.6 Practicality and Applications

One obvious objection to PUF ROKs is that they assume shared access to the resource. In many instances, this assumption does not hold. However, as we will describe in Section 5, we have implemented a proof-of-concept PUF ROK on a small portable device that measures 44×60 mm. Based on this experience, we find that it would be quite reasonable to integrate PUF ROK functionality into devices similar to USB thumb drives. Clearly, such a device could be passed between users who are generally in close proximity.

For remote users, a more complicated structure would be needed. Specifically, a PUF ROK for remote use would function in a manner similar to a TPM. That is, Bob's TPM-like PUF ROK would generate a public key that Alice would use for encryption. Later, Bob's device would generate the corresponding private key to decrypt the message. Clearly, Alice would need assurance that the public key actually came from a PUF ROK. Unfortunately, there is no easy solution to this. Instead, we find the most straightforward approach to

be exactly that used by TPMs. That is, the PUF ROK device would require a certificate created by the manufacturer, storing a persistent private key generated by the manufacturer. This key would then be used to sign all PUF ROKs from that device, and Alice could confirm the signature with the manufacturer's public key.

In short, our PUF ROK device infrastructure for remote users would mirror TPM behavior. However, there is one major exception: The device is trusted to enforce the behavior that the PUF ROK can only be used once. This behavior does not exist in TPMs. However, PUF ROKs could clearly be integrated into any custom SoC design that is functionally similar to a TPM.

Now consider the applications of PUF ROKs. Goldwasser *et al.* [18] proposed a technique for one-time programs, based on the assumption that the application is encrypted under a one-time use key. Closely related to one-time programs are delegated signatures. If Alice has a persistent key sk_A , she could encrypt this key with the PUF ROK as $e(sk_A)$. Bob would then provide this encrypted key to the PUF ROK, which decrypts it and uses the decrypted key to sign a single document on Alice's behalf.

PUF ROKs could also be used to generate self-destructing messages. If Alice has a portable PUF ROK device, she could use it to generate `Enc(m)`. After receiving the message, Bob could use the device to decrypt the message. Once this is done, repeated attempts to decrypt the message would fail, as the Reg would no longer store the necessary challenge input.

Finally, consider the scenario of usage control. In this case, Bob has a public key PUF ROK device that contains the TPM-like endorsement key EK. Bob could use the device to retrieve the signed pk , which he sends to Alice. Alice, after confirming the signature, uses the key to encrypt the protected resource, sending the result to Bob. Bob can then use the sk stored on the PUF ROK to access the resource. Once the CC uses sk , this key is no longer accessible, and access to the resource is revoked.

5. IMPLEMENTATION

To demonstrate a proof-of-concept for our PUF ROK design, we developed a prototype implementation. As the design requires a combination of hardware and software, we desired a platform that would be advantageous for both pieces of development. Our solution was to use the KNJN Saxo-L development board [4]. This board features an Altera Cyclone EP1C3T100 field-programmable gate array (FPGA) alongside an NXP LPC2132 ARM processor. The FPGA and ARM are directly connected via a Serial Peripheral Interface (SPI). The board also offers a USB-2 adaptor, in addition to the JTAG adaptor that is commonly used for FPGA development. In addition, the form factor of the board measures 44×60 mm, making it highly portable.

In our prototype development, we chose to use the FPGA only for components that require a hardware implementation. Specifically, we implemented a small PUF and register on the FPGA to capture the essence of the feedback loop. All other portions, including the error-correcting unit (ECU) and the crypto core (CC) were implemented in software on the ARM processor. Figure 4 shows the high-level layout of our implementation on the KNJN board.

Our PUF design consisted of 32 1-bit ring oscillator PUFs, as shown in Figure 1. Each of these circuits consisted of a

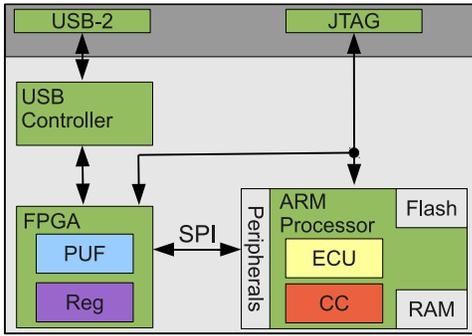


Figure 4: Basic hardware layout of a PUF ROK implemented on the KNJN Saxo-L development board

ring oscillator constructed from 37 inverting gates. In our experiments, we found that using fewer than 37 gates yielded less consistency in the PUF behavior. That is, smaller PUFs increase the number of bit errors that must be corrected. The output from the ring oscillators was linked to 20-bit counters that were controlled by a 16-bit timer. The timer was synchronized with a 24 MHz clock, indicating that the timer would expire (as a result of an overflow) after 2.73 ms. When the timer expires, the values in the counters are compared, producing a 1 or 0 depending on which counter had the higher value. This design used 2060 of the 2910 (71%) logic cells available on the FPGA. Each execution of the PUF produced 32 bits of output. Consequently, to generate larger keys, the ARM processor polled the PUF multiple times, caching the result until the key size was met.

To put the performance of the PUF into perspective, we compared the execution time with measurements [2] reported by NXP, the device manufacturer. Some of NXP’s measurements are reported in Figure 5. As each PUF execution (producing 32 bits of output) requires 2.73 ms to overflow the timer, it is slower than encrypting one kB of data in AES. Observe, though, that larger PUFs would still only require 2.73 ms. Consequently, the overhead of executing the PUF can remain small, especially as large amounts of data are encrypted or decrypted.

Symmetric Algorithm	Time (ms/kB)	RSA Operation	Time (s)
AES-CBC	1.21	1024-bit encrypt	0.01
AES-ECB	1.14	1024-bit decrypt	0.27
3DES-CBC	3.07	2048-bit encrypt	0.05
3DES-ECB	3.00	2048-bit decrypt	2.13

Figure 5: NXP cryptographic measurements

The comparison the RSA encryption and decryption is stark. Observe that the 2.73 ms required to execute the PUF is 27.3% of the time to perform a 1024-bit encryption in RSA. As the key size increases (assuming the PUF size is increased accordingly so that only one polling is needed), the PUF execution time becomes 0.13% overhead for 2048-bit RSA decryption. Thus, the performance impact of polling the PUF during key generation is minimal.⁶

⁶Obviously, there is additional work required to convert the PUF output into a usable key. However, the precise timing of this work is implementation-dependent, and the al-

Regardless of the size of the PUF used, one challenge that is unavoidable is random inconsistencies in the PUF output. Specifically, a small number of bit errors will inevitably occur as a result of the randomness inherent to PUFs. To counteract this behavior, we employed Reed-Solomon (RS) error-correcting codes [25]. Specifically, we adapted the Rockliff’s implementation [32] in our prototype.

RS(n, k) codes operate as follows. A string of k symbols, each m bits in length, is concatenated with an $n - k$ syndrome, where $n = 2^m - 1$. Based on this syndrome, when the input is used again, the codes can correct up to $(n - k)/2$ corrupted symbols. In our implementation, we used RS(15, 9) codes, as the PUF output each time was 32 bits. Observe that $m = 4$ in this code, so this $k = 9$ is a sufficient size, as there would be nine 4-bit symbols (a total of 32 bits). Furthermore, when the PUF is polled later, this code can account for three corrupted symbols (potentially up to 12 bits in the PUF). However, in our experiments, the average Hamming distance between the original PUF response and later responses was 0.1, with a maximum of two bit errors that occurred in one execution. Clearly, this code is sufficient for our implementation.

In our implementation, we adapted the PolarSSL cryptographic library [1] for execution on the ARM processor. This open source library is designed to be small enough for use in embedded devices. The LPC2132 model offers only 16 kB of RAM and 64 kB of Flash memory. As this is quite a small amount of storage space, the library actually would not fit in the device’s available memory. Specifically, the library contains a number of I/O functions that are not suited for our implementation. Consequently, we customized the code by removing this unused functionality and were able to make the code fit within the confines of the device.

Based on the experience of building this prototype, we offer the following insights for creating production-quality PUF ROKs. First, employing both an FPGA and an ARM processor adds to the complexity of the system design. As an alternative approach, one could leverage ARM softcore designs and place them within the FPGA itself. This would simplify the circuitry on the board itself.

Additionally, our current design is not optimal for production-quality PUFs. Specifically, it creates a one-to-one correlation between a single bit in the input challenge C_i and the corresponding response bit in R_i . As such, if C_i and C_j differ by only a single flipped bit, then R_i and R_j will also differ by only the same flipped bit. To prevent this correlation and produce behavior that more closely resembles an ideal PUF, the circuit design should randomly select pairs from a pool of ring oscillators, rather than having persistent pairs. As this issue was addressed in [36], interested readers should consult that work for more information.

Finally, our work used a resource-limited development board. Specifically, 2910 logic cells is considerably smaller than most FPGAs (*e.g.*, the Xilinx Spartan-3E has 10,476). Also, recall that the amount of memory available was too small to hold a cryptographic library that is *intended* for embedded devices. Consequently, we feel confident that our architecture could be easily adapted to larger devices, even as if the size of the PUF is increased to produce larger keys.

gorithms typically employed are significantly more efficient than the modular exponentiation. As such, we focus solely on the PUF measurement in our analysis.

6. SECURITY ANALYSIS

For our security analysis, we consider the case of a probabilistic polynomial-time (PPT) attacker \mathcal{A} , with two goals. First, the goal of \mathcal{A} is to recover just the key used to encrypt or decrypt a single message. The second goal considered is to model the PUF, which would enable the attacker to emulate the PUF ROK in software, thereby negating the hardware ROK guarantee. Initially, in both cases, we assume the adversary is capable of (at most) eavesdropping on bus communication. That is, the adversary is unable to observe communication between the cores in the SoC design.

Under this model, \mathcal{A} is able to observe the data passing between the PC and memory, or between the PC and a network. Observe, though, that these messages consist exclusively of the plaintext m and the encrypted $e(m)$. Thus, the attack is a known-plaintext attack. However, this information offers no additional knowledge to \mathcal{A} . Even if \mathcal{A} managed to reconstruct the key \mathcal{K} (with negligible probability under the PPT model), this key is never used again.

The only use of reconstructing \mathcal{K} in this manner is to attempt to reverse engineer the behavior of the PUF. However, recall that our design involved hashing the PUF output when creating the keys. Consequently, $\mathcal{K} = H(R_i)$, where H is a robust cryptographic hash function. As a result, \mathcal{A} again has only a negligible probability of reconstructing R_i . Yet, we can take this analysis even further, because R_i by itself is useless. That is, \mathcal{A} would also need to know the corresponding C_i (or R_{i+1}) to begin to model the PUF. Thus, \mathcal{A} would have to accomplish a minimum of *four* feats, each of which has only a negligible probability of occurring. Thus, we do not find such an attack to be feasible.

To continue the analysis, we loosen our assumed restrictions and grant \mathcal{A} the ability to probe inside the SoC and observe all data transferred between the cores. Clearly, such an adversary would succeed, as the data passed between the PUF and the CC occurs in the open. However, this attack model is so extreme that only the most dedicated and motivated adversaries would undertake such a task. Similarly, users who are faced with such powerful adversaries are likely to have extensive resources themselves. Thus, these users are likely to shield the processor using known tamper-resistance techniques, and we find this threat to be minimal.

Moving away from the PPT model, we can return to the discussion of fault injection [11, 10, 9, 8, 30] and freezing [5] attacks. Fault injection attacks fail to threaten the confidentiality of the system, because these attacks are based on repeatedly inducing the fault with the same key. However, PUF ROKs can only be used once. At best, a fault injection would become a denial-of-service, as the key would not correctly encrypt or decrypt the message. Freezing attacks are similarly unsuccessful, because they operate on the assumption that the key existed in addressable memory at some point. However, that is not the case with PUF ROKs. These keys are generated dynamically and are never explicitly stored outside the processor itself. Thus, PUF ROKs offer robust defenses against these physical attacks.

One final class of attacks to consider is power analysis [26]. Simple power analysis (SPA) involves monitoring the system's power fluctuation to differentiate between portions of cryptographic algorithms. This information leakage can reveal how long, for instance, a modular exponentiation takes, which reveals information about the key itself. Differential power analysis (DPA) observes the power fluctuations over

time by repeatedly executing the cryptographic algorithm *with the targeted key*. Ironically, DPA is considered harder to defend against than SPA. And yet, PUF ROKs are immune to DPA (since repeated execution is not allowed) while vulnerable to SPA. Even though SPA is a potential threat, known techniques can prevent these attacks [37].

7. CONCLUSION

In conclusion, this work presents a novel hardware-based approach to generating read-once keys (ROKs). Our underlying strategy is to integrate a PUF with a register to create a feedback loop. The result is that no data required for the PUF ROK ever exists outside of the processor itself. In addition, the feedback loop continuously overwrites the contents of the register, thereby destroying the key immediately upon use. As such, the design successfully captures the notion of a ROK.

In this work, we have defined a ROK in terms similar to a Turing machine. We presented our architectural design and proved that it matches the formalism. We described applications of PUF ROKs and addressed concerns regarding their practicality and usability. We presented details of our prototype design and shared insights regarding future production-quality implementations. We presented a security analysis of PUF ROKs under the PPT adversary model, and we also demonstrated that PUF ROKs are resilient against even more powerful adversaries with the ability to perform physical attacks on the device. In summary, we have demonstrated that PUF ROKs are both feasible and secure.

8. REFERENCES

- [1] Polarssl: Small cryptographic library. <http://www.polarssl.org/>, 2008.
- [2] Encryption for ARM MCUs. http://ics.nxp.com/literature/presentations/microcontrollers/pdf/nxp_security_innovation_encryption.pdf, 2010.
- [3] Ironkey military strength flash drives. <http://www.ironkey.com/>, 2010.
- [4] KNJN FPGA development boards. <http://www.knjn.com/FPGA-FX2.html>, 2010.
- [5] A. Akavia, S. Goldwasser, and V. Vaikuntanathan. Simultaneous hardcore bits and cryptography against memory attacks. In *TCC '09: Proceedings of the 6th Theory of Cryptography Conference on Theory of Cryptography*, pages 474–495, Berlin, Heidelberg, 2009. Springer-Verlag.
- [6] M. J. Atallah, E. D. Bryant, J. T. Korb, and J. R. Rice. Binding software to specific native hardware in a VM environment: The PUF challenge and opportunity. In *VMSEC '08*. ACM, 2008.
- [7] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2001.
- [8] A. Berzati, C. Canovas, J.-G. Dumas, and L. Goubin. Fault attacks on RSA public keys: Left-to-right implementations are also vulnerable. In *CT-RSA '09: Proceedings of the The Cryptographers' Track at the RSA Conference 2009 on Topics in Cryptology*, pages 414–428, Berlin, Heidelberg, 2009. Springer-Verlag.

- [9] A. Berzati, C. Canovas, and L. Goubin. In (security) against fault injection attacks for CRT-RSA implementations. *Fault Diagnosis and Tolerance in Cryptography, Workshop on*, 0:101–107, 2008.
- [10] A. Berzati, C. Canovas, and L. Goubin. Perturbating RSA public keys: An improved attack. In E. Oswald and P. Rohatgi, editors, *Cryptographic Hardware and Embedded Systems (CHES 2008)*, volume 5154 of *Lecture Notes in Computer Science*, pages 380–395. Springer Berlin / Heidelberg, 2008.
- [11] E. Brier, B. Chevallier-mames, M. Ciet, C. Clavier, and École Normale Supérieure. Why one should also secure RSA public key elements. In *Cryptographic Hardware and Embedded Systems (CHES 2006)*, volume 4249 of *Lecture Notes in Computer Science*, pages 324–338. Springer-Verlag, 2006.
- [12] B. Danev, T. S. Heydt-Benjamin, and S. Čapkun. Physical-layer identification of RFID devices. In *Proceedings of the USENIX Security Symposium*, 2009.
- [13] S. Devadas, E. Suh, S. Paral, R. Sowell, T. Ziola, and V. Khandelwal. Design and implementation of PUF-based “unclonable” RFID ICs for anti-counterfeiting and security applications. In *2008 IEEE International Conference on RFID*, pages 58–64, 2008.
- [14] K. B. Frikken, M. Blanton, and M. J. Atallah. Robust authentication using physically unclonable functions. In *Information Security Conference (ISC)*, September 2009.
- [15] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas. Controlled physical random functions. In *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC)*, 2002.
- [16] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas. Silicon physical random functions. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS '02)*, 2002.
- [17] R. Geambasu, T. Kohno, A. Levy, and H. M. Levy. Vanish: Increasing data privacy with self-destructing data. In *Proc. of the 18th USENIX Security Symposium*, 2009.
- [18] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. One-time programs. In *CRYPTO 2008: Proceedings of the 28th Annual conference on Cryptology*, pages 39–56, Berlin, Heidelberg, 2008. Springer-Verlag.
- [19] V. Goyal, Y. Ishai, A. Sahai, R. Venkatesan, and A. Wadia. Founding cryptography on tamper-proof hardware tokens. In D. Micciancio, editor, *Theory of Cryptography*, volume 5978 of *Lecture Notes in Computer Science*, pages 308–326. Springer Berlin / Heidelberg, 2010.
- [20] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls. FPGA intrinsic PUFs and their use for IP protection. In *Proceedings of the 9th Cryptographic Hardware and Embedded Systems Workshop (CHES)*, pages 63–80, 2007.
- [21] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls. Physical unclonable functions and public-key crypto for FPGA IP protection. In *International Conference on Field Programmable Logic and Applications*, pages 189–195, 2007.
- [22] M. Kirkpatrick and E. Bertino. Physically restricted authentication with trusted hardware. In *The Fourth Annual Workshop on Scalable Trusted Computing (ACM STC '09)*, November 2009.
- [23] M. S. Kirkpatrick and S. Kerr. Enforcing physically restricted access control for remote data. In *1st ACM Conference on Data and Application Security and Privacy (CODASPY)*, February 2011.
- [24] M. S. Kirkpatrick, S. Kerr, and E. Bertino. PUF ROKs: Generating read-once keys with physically unclonable functions (extended abstract). In *6th Annual Cyber Security and Information Intelligence Research Workshop (CSIIRW)*, April 2010.
- [25] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, pages 203–209, 1987.
- [26] P. Kocher, J. Jaffe, and B. Jun. Introduction to differential power analysis and related attacks. Technical report, Cryptography Research, 1998.
- [27] V. Kolesnikov. Truly efficient string oblivious transfer using resettable tamper-proof tokens. In *TCC*, pages 327–342, 2010.
- [28] K. Lofstrom, W. Daasch, and D. Taylor. IC identification circuit using device mismatch. In *Solid-State Circuits Conference, 2000. Digest of Technical Papers. ISSCC. 2000 IEEE International*, pages 372–373, 2000.
- [29] S. Narayanan, A. Raghunathan, and R. Venkatesan. Obfuscating straight line arithmetic programs. In *DRM '09: Proceedings of the ninth ACM workshop on Digital rights management*, pages 47–58, New York, NY, USA, 2009. ACM.
- [30] A. Pellegrini, V. Bertacco, and T. Austin. Fault-based attack of RSA authentication. In *Design Automation and Test in Europe (DATE)*, March 2010.
- [31] M. Riley and I. Richardson. Reed-solomon codes. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/pscico-guyb/realworld/www/reedsolomon/reed_solomon_codes.html, 1998.
- [32] S. Rockliff. The error correcting codes (ecc) page. <http://www.eccpage.com/>, 2008.
- [33] L. F. G. Sarmanta, M. van Dijk, C. W. O'Donnell, J. Rhodes, and S. Devadas. Virtual monotonic counters and count-limited objects using a tpm without a trusted os. In *STC '06: Proceedings of the first ACM workshop on Scalable trusted computing*, pages 27–42, New York, NY, USA, 2006. ACM.
- [34] G. E. Suh and S. Devadas. Physical unclonable functions for device authentication and secret key generation. In *Proceedings of the 44th IEEE Design Automation Conference (DAC)*, pages 9–14. IEEE Press, 2007.
- [35] G. E. Suh, C. W. O'Donnell, and S. Devadas. AEGIS: A single-chip secure processor. In *Elsevier Information Security Technical Report*, volume 10, pages 63–73, 2005.
- [36] G. E. Suh, C. W. O'Donnell, and S. Devadas. Aegis: A single-chip secure processor. *IEEE Design and Test of Computers*, 24(6):570–580, 2007.
- [37] V. Sundaresan, S. Rammohan, and R. Vemuri. Defense against side-channel power analysis attacks on microelectronic systems. pages 144–150, Jul. 2008.