PLayPUF: Programmable Logically Erasable PUFs for Forward and Backward Secure Key Management

Chenglu Jin[†], Xiaolin Xu[‡], Wayne Burleson[‡], Ulrich Rührmair[§] and Marten van Dijk[†]

[†]University of Connecticut – chenglu.jin@uconn.edu,vandijk@engr,uconn.edu
[‡]University of Massachusetts Amherst – xiaolinx@umass.edu, burleson@ece.umass.edu
[§]Horst Görtz Institute for IT-Security, Ruhr Universität Bochum – ruehrmair@ilo.de

October 28, 2015

Abstract

A silicon Physical Unclonable Function (PUF) is a hardware security primitive which implements a unique and unclonable function on a chip which, given a challenge as input, computes a response by measuring and leveraging (semiconductor process) manufacturing variations which differ from PUF to PUF. In this paper, we observe that by equipping a PUF with a small, constant-sized, tamper-resistant state, whose content cannot be modified, but can be read by adversaries, new and powerful cryptographic applications of PUFs become feasible. In particular, we show a new hardware concept which we call a Programmable Logically erasable PUF (PLayPUF). Its distinctive feature is that it allows the selective erasure of single challenge-response pairs (CRPs) without altering any other PUF-CRPs. The selective erasure of a CRP can be programmed a-priori by using a counter to indicate how many times the CRP can be read out before erasure.

We show PLayPUFs can realize forward and *backward* secure key management schemes for public key encryption. The new notion of backward security informally means that even if an attacker uncovers a session key through the key management interface, the legitimate user will detect this leakage before he will ever use the session key. Backward security and its implementation via PLayPUFs allow the construction of novel, self-recovering certificate authorities (CAs) without relying on a digital master key. Our new CAs immediately detect key exposure through their interfaces, and recover from it without stopping their service, and without ever issuing certificates based on such exposed keys. This is a crucial step forward in implementing secure key management. We deliver a full proof-of-concept implementation of our new scheme on FPGA together with detailed performance data, as well as formal definitions of our new concepts, including the first definition of stateful PUFs.

Contents

1 Introduction

A Physical Unclonable Function (PUF) is a hardware security primitive that utilizes process manufacturing variation to implement a unique unclonable function. It has been shown that just having PUFs (with no additional properties and not assuming "anything else" like some other computational hardness assumption) is not sufficient to construct a secure Key Exchange (KE) or Oblivious Transfer (OT) protocol in the PUF re-use model (where used PUFs at some moment in the future may end up at malicious parties) [1]. It has been observed that by using erasable PUFs instead secure KE and OT protocols can be constructed [2]. Erasable PUFs have an additional interface property which allows one to erase (deny access to) challenge response pairs without affecting others. Implementing this efficiently based on physics principles alone is an open problem. Therefore, in this paper we introduce logically erasable PUFs where erasability is implemented by the PUF interface using $O(\lambda)$ tamper-resistant state, where λ is a security parameter. Since the Trusted Computing Base (TCB) is now slightly extended to include this extra tamper-resistant state and tamper-resistant control circuitry (for implementing the erasability functionality), the control circuitry can be used to implement more intelligence in the PUF interface so that erasability of challenge response pairs can be programmed allowing one to set a maximum number of times a challenge response pair can be read out before it will be erased.

This new concept of a Programmable Logically Erasable PUF, coined PLayPUF[1], can be used to efficiently implement a forward [3] and "backward" secure secret key management scheme for public key encryption without having to use some secret master key: During initialization session keys are stored in an obfuscated way with the property that only a certain specific programmable logically erasable PUF must be used to extract session keys. Our concept of backward security means here that after this initialization, if a session key is leaked through the PUF interface to a party other than the legitimate owner, then the owner detects this leakage *before* he will ever use this session key (in which case he will generate a new public key with new session keys).

A PLayPUF can also be used to construct a forward and backward secure digital signature scheme; this allows us to design a Certificate Authority (CA) without a digital master key which can automatically recover from key exposure through its "key storage interface". In particular, our approach allows the CA to detect such key exposure immediately before usage of exposed keys after which it regenerates a new public key and secret key sequence to replace the compromised one. No fake certificates based on exposed keys will pass verification. Our PLayPUF applications are crucial steps forward in implementing secure key management.

1.1 Contributions

1. We introduce a formal definitional framework for PUFs with stateful interfaces in Section 2 and discuss in detail the Trusted Computing Base (TCB) of such physical objects.

2. Our definitional framework allows us to introduce and rigorously define Programmable Logically Erasable PUFs (PLayPUFs) in Section 3, where we also describe a PLayPUF design together with its FPGA implementation and evaluation.

3. We define our concept of "backward" security and show how to construct a forward and "backward" secure key management scheme and a self-recovering certificate authority based on a PLayPUF in Section 4.

[1] Represents a malleable substance that can be shaped to one's own desires, similar to play dough for toddlers.

Related works are detailed throughout the paper.

2 Definitional Framework for PUFs

Intuitively, a silicon Physical Unclonable Function (PUF) is a fingerprint of a chip, that (1) leverages process manufacturing variation to generate a unique function taking "challenges" as input and generating "responses" as output, which (2) cannot be cloned in hardware (the PUF's internal behavior, e.g. its unique physical characteristics or behavior of its wires, cannot be read out accurately enough; also it is not feasible to manufacture two PUFs with the same responses to a significant subset of challenges) and (3) cannot be efficiently learned given a "polynomial number" of challenge response pairs (making it impossible to impersonate/clone the function's behavior to a new random challenge in software).

A formal (ideal) definition of a PUF (we give a detailed explanation afterwards) is as follows:

Definition 1. [Stateless PUFs] *A family of stateless PUFs is described by manufacturing processes* $\{M_\lambda\}$ *such that each physical object* $P_\lambda \leftarrow M_\lambda$ *can be stimulated by challenges* c *from a challenge space* C_λ*, by which it reacts with corresponding responses* r *from a response space* R_λ*. We reserve a special symbol* $\perp \in R_\lambda$ *for the case, where if stimulated by a challenge* $c \in C_\lambda$ *for which* P_λ *does not have a response, then it can output* $\perp$*. We model* P_λ *by an associated function* $g[P_\lambda] : C_\lambda \rightarrow R_\lambda$ *that maps challenges* c *to responses* $r = g[P_\lambda](c)$*; functions* $g[P_\lambda]$ *execute in* $poly(\lambda)$ *time. The pairs* $(c, g[P_\lambda](c))$ *are called challenge-response pairs (CRPs) of PUF* P_λ*. The manufacturing process* M_λ *can be viewed as a distribution from which a function* $g[P_\lambda]$ *is drawn.*

Parameter λ *represents a design parameter of* M_λ *and characterizes the "unclonability" property: Given any* $poly(\lambda)$ *sized list of CRPs* $(c_i, g[P_\lambda](c_i))$ *(intuitively, the CRPs collected by an attacker by, e.g., eavesdropping, stealing, or black-box access to* P_λ *while in possession of the PUF) it is impossible to determine a response* $g[P_\lambda](c)$ *for a* $c \notin \{c_i\}$ *by using a probabilistic* $poly(\lambda)$ *time algorithm with probability* $> 2^{-\lambda}$*:*

$$\forall_\lambda \ \forall_{c \leftarrow C_\lambda} \ \forall_{ppt\ \mathcal{A}} \ Prob_{g \leftarrow M_\lambda}[g(c) = r \mid \perp \neq r \leftarrow \mathcal{A}^{G_{g,c}, M_\lambda}(1^\lambda, c)] \leq 2^{-\lambda},$$

where $\mathcal{A}$ *has oracle access to (1)* $G_{g,c}$ *which outputs* $g(c')$ *for inputs* $c' \in C_\lambda$ *with* $c' \neq c$*, and halts on input* c*, and to (2) distribution* M_λ*.*

Measurement Noise. Definition 1 does not model measurement noise in the PUF itself. In practice one may need error correction or a fuzzy extractor to correct noise [4]. Also, notice that in practice responses for challenges that are close in some metric (e.g., Hamming distance) are correlated and Definition 1 must be slightly weakened by redefining oracle $G_{g,c}$ to output $g(c')$ for inputs $c' \in C_\lambda$ with $d(c', c) > t_\lambda$, and to output $\perp$ on inputs $c' \in C_\lambda$ with $d(c', c) \leq t_\lambda$, where $d(.,.)$ is some metric (e.g., Hamming distance) and t_λ some threshold. Note that such an oracle is not needed if the physical object corresponding to g includes interface logic which expands each input challenge into a larger sized challenge in a code $\mathcal{C}$ with minimum distance at least t_λ, and uses the larger sized challenge as input for the functionality within the physical object that computes on manufacturing variations. We assume that this type of interface logic as well as interface circuitry for fuzzy extraction or error correction is included in the physical objects, and therefore we model combined PUFs + interfaces (which Definition 1 calls PUFs) as ideal functionalities without noise and with uncorrelated responses.

Manufacturing Variations and Hardware Unclonability. Definition 1 does not mention manufacturing variation as the source for unclonability at all and allows physical objects that implement digital circuitry without any manufacturing variations, e.g., a digital PUF [5] which has a fused secret key K and implements $g(c) = Enc_K(c)$ for some semantically secure encryption scheme. In general, however, the physical objects are a combination of two kinds of functionalities: (1) a function that depends on the manufacturing variations during fabrication of the object, and (2) a function that does not depend on manufacturing variations. In the sequel, if we mean the first we say "PUF" and if we mean the latter we say "interface"; their distinction is fuzzy, however, as manufacturing variations are exploited by analog computing, a translation needs to be made from analog to digital and does the conversion functionality belong to the interface or not? Examples of more clear interface functionalities are circuitry that corrects measurement noise, or circuitry that expands an input challenge into a chain of challenges that are each fed to the PUF in order to increase the size of the response.

The adversary with physical access to the PUF + interface may use side channel analysis, electron microscopy, photonic emission analysis, etc. to figure out what signals are being propagated inside the PUF + interface; this would lead to microscopic measurements of the manufacturing variations that define the PUF and can be used to either create a SW clone or to duplicate the manufacturing variations in a HW clone. HW unclonability not only means that we believe that manufacturing variations cannot be duplicated in HW, not even by the manufacturer himself, but we also believe the stronger assumption that collection of useful information through physical attacks (which is needed to create a HW clone in the first place) is not possible (in order to validate this assumption, some appropriate countermeasures need to be implemented against side channel analysis [6]): I.e., we believe that physical access does not give an adversary any advantage over an adversary with black box access to the physical object. By only considering an adversary to only having black-box access, we implicitly assume this strong form of HW unclonability. This is formalized in Definition 1 where the adversarial algorithm $\mathcal{A}$ has adaptive access to an oracle which represents the physical object as a black box.

Trusted Computing Base (TCB). The impossibility of collecting useful information through physical attacks implies that all of the PUF and interface should remain *tamper-resistant* in the sense that an adversary cannot purposefully modify functionality into something that allows him to gain an advantage over an adversary with only black-box access, e.g. he cannot purposefully modify the interface functionality and cannot separate the bonding between PUF and interface. In addition, the internals of those parts of the interface circuit and PUF, which if leaked to an adversary would gain him an advantage over the adversary with only black-box access, should remain *private*. The TCB of a PUF is characterized by the parts of the PUF that are

- tamper-resistant and private,
- tamper-resistant and public, or
- public without tamper-resistance.

Outside the TCB is an interface part of the PUF which is public and which does not need to be tamper-resistant. This means that we should refine Definition 1: Oracle $G_{g,c}$ only represents black-box access to the TCB of the physical object. Oracle $G_{g,c}$ interacts with the public part of the physical object which is under control of adversary $\mathcal{A}$ (who can modify its functionality in whatever malicious way as desired). So, oracle access is now a protocol between $\mathcal{A}$ simulating the public part

of the physical object and oracle $G_{g,c}$ modeling the TCB part of the physical object as a finite state machine. The final challenge-response functionality $g(.)$ is given by the very first input to the oracle which represents the challenge and the very last output (after all interactions) of the oracle which represents the response. In the remainder of this paper we assume such an extension to Definition 1 although we never make this explicit.

Belief in the stronger HW unclonability (as described above) reduces to belief in the described TCB: As the TCB of the physical object is larger, it physically embodies a larger attack surface increasing the probability of some implementation/fabrication weakness which may allow the adversary to gain useful information by means of a physical attack. Therefore, we want physical objects to be small in size (which also reduces the cost of their fabrication).

With respect to the TCB, Definition 1 excludes $\mathcal{A}$ in getting black-box access to other PUFs; we know that adversaries have access to multiple PUFs but if such access does not help in attacking a specific PUF, then we do not need to model this. Instead we may model the manufacturing process M_λ as a distribution from which functionalities $g[P_\lambda]$ are drawn, i.e., all drawn functionalities are independent and identically distributed, hence, access to multiple PUFs does not help the adversary other than learning the general statistics of distribution M_λ. The latter is modeled by giving $\mathcal{A}$ oracle access to M_λ.

Finally, Definition 1 implicitly assumes that manufacturing is trusted as adversary $\mathcal{A}$ cannot maliciously change M_λ. In particular, he cannot create malicious or bad PUFs [1]. This is realistic in those scenarios where e.g. the trusted manufacturer collects a list of CRPs to verify authenticity later on as a trusted third party (however, this is not a scalable solution) or if PUFs are used in protocols where the PUF owner is not the adversary and is therefore trusted to have bought the PUF directly from the trusted manufacturer.

Software Unclonability. As explained above Definition 1 implicitly assumes HW unclonability in the sense of adversaries with only black-box access to the physical object; it explicitly states that the object can also not be cloned in software, i.e., there does not exist a polynomial time algorithm which can predict a response for a new challenge c (whose response has not yet been given by the oracle) of its choice with probability $> 2^{-\lambda}$. In other words, new responses have at least λ bits entropy unknown to attackers (Definition 1 does allow some of the response to be SW cloned as long as at least λ bits entropy are guaranteed).

Notice that we consider polynomial time adversaries meaning that the PUF may use some computational hardness assumption in order to achieve unclonability. This assumption may be implicit in the PUF design because of how manufacturing variations are combined. But it can also be explicitly used as in e.g. assuming a semantically secure encryption scheme in a digital PUF [5].

Other Definitions. In this paper we do not consider protocols based on PUFs: a formal treatment of how to reason about physical transfers of PUFs in protocols is given in [7] and is complementary to our framework.

2.1 Stateless PUFs vs Stateful PUFs

In practice a stateless PUF can be used in many different scenarios by exploiting its uniqueness and unclonability. For instance, similar to a fingerprint of a human, it can help us identify chips. Also, key generation is a promising application of PUF technology because the secret key can be stored in a physical form instead of a digital form, meaning that the device must be powered on while the attacker is attempting to extract secret key information from it.

Meanwhile many cryptographic protocols based on stateless PUFs have been proposed, including Key Exchange (KE), Oblivious Transfer (OT) and Bit Commitment (BC) protocols [7–9]. These protocols target real-world applications in the so-called proper PUF model where PUFs are destroyed after its protocol usage. However, in practice we want to re-use PUFs and this creates new exploitable vulnerabilities at the protocol level, which does not require cloning or characterizing the PUF itself [1]. This PUF re-use model has been shown to influence all the existing PUF-based KE and OT protocols [1]. In particular, the impossibility of KE (and consequently OT) based on a family of stateless PUFs and without "other" security assumptions was proven in [2], see also [10]. Therefore, in order to be able to design more advanced secure crypto protocols based on PUFs, we either need to allow computational hardness assumptions or allow state.

Of course in practice we do allow computational hardness assumptions based on which efficient secure KE exists without the need for (physical transmission of) PUFs. Nevertheless the above discussion shows the limitations of stateless PUFs: In order to understand how hardware security in the form of PUFs can play a role in implementing novel security functionalities (besides KE etc.) based on a small TCB, we investigate PUFs that allow tamper-resistant state, i.e., non-volatile memory (which can be read but cannot be modified by the attacker). In this paper we show that a small stateful extension to a PUF's TCB efficiently implements so-called erasability of PUF responses. It allows two advances in key management: implementation of backward security and self-recovering certificate authority, see Section 4.

Assuming non-volatile memory to be private is, given existing physical attacks, not realistic. For this reason, we may only assume tamper-resistant non-volatile memory in PUFs with state. Therefore, belief in the above stronger HW unclonability intuitively means a TCB in which state is not required to be private. PUFs with tamper-resistant not-private state are defined as follows:

Definition 2. [Stateful PUFs] *A family of PUFs with tamper-resistant state is described by manufacturing processes $\{M_\lambda\}$ such that each physical object $P_\lambda \leftarrow M_\lambda$ has an internal state s from a state space S_λ of $O(\lambda)$ size, and can be stimulated by challenges c from a challenge space C_λ, by which it reacts with corresponding responses r from a response space R_λ (which includes the empty response $\perp$). We model P_λ by an associated poly(λ) time algorithm $g[P_\lambda] : C_\lambda \times S_\lambda \to R_\lambda \times S_\lambda$ that maps challenges c to responses r with*

$$(r, s_{new}) = g[P_\lambda](c, s_{current}), \tag{1}$$

where initially $s_{current} = \epsilon \in S_\lambda$. The manufacturing process M_λ can be viewed as a distribution from which an algorithm $g[P_\lambda]$ is drawn.

Parameter λ represents a design parameter of M_λ and characterizes the "unclonability" property: For all λ, $c \in C_\lambda$, and ppt algorithms $\mathcal{A}$,

$$Prob_{g \leftarrow M_\lambda}[\exists_{s_{current}, s_{new} \in S_\lambda} \ (r, s_{new}) = g(c, s_{current}) \mid \perp \neq r \leftarrow \mathcal{A}^{G_{g,c}, M_\lambda}(1^\lambda, c)] \leq 2^{-\lambda},$$

where oracle $G_{g,c}$ keeps state $s_{current}$ (initialized to $s_{current} = \epsilon$) and receives besides input $c' \in C_\lambda$ a second input $k \in \{0, 1\}$ indicating whether $\mathcal{A}$ wants to receive the output of the oracle:

If $c' \neq c$ or $k = 0$, then $G_{g,c}$ simulates algorithm $(r, s_{new}) = g(c', s_{current})$, outputs (r, s_{new}) if $k = 1$ and outputs the empty string if $k = 0$, and updates its state $s_{current}$ to s_{new}; if $c' = c$ and $k = 1$, then oracle $G_{g,c}$ halts.

In the above definition flag $k = 0$ indicates that the adversary is not in possession of the PUF while it is being challenged. In this case the adversary does not receive the response. Since challenges

are in general used as public strings in protocols or systems, we assume that the adversary does learn the sequence of challenges issued to the PUF when it was not in his possession. For this reason the oracle is fine with processing challenge c (by updating state $s_{current}$) if $k = 0$ as it will not reveal the corresponding response. If $k = 1$ (indicating that the PUF is in possession of the adversary), then $\mathcal{A}$ indeed learns from the oracle how state $s_{current}$ is updated (it is not private), however, he can not modify its content (it is tamper-resistant). The adversary's task is to predict a non-empty response r with $(r, .) = g(c, s)$ for some s without asking the oracle for response r: this should be hard and represents SW unclonability.

Definition 2 is very strong in that in theory it allows secure processors: $s_{current}$ can be used to implement a memory integrity checking interface [11] and an ORAM interface [12] (with their data structures in external DRAM in the public part/domain), to implement certified execution etc., in fact the Aegis secure processor architecture was extended with PUFs for storing secret keys in [13]. In practice, a processor should execute with small performance overhead and this means that it needs an internal cache structure with the memory integrity checking interface and ORAM interface between the lowest level cache and DRAM. Hence, $s_{current}$ should include this cache structure which should remain private (besides being tamper-resistant). The TCB of a secure processor becomes quite large in area size and must include private non-volatile memory. For this reason literature in secure processor technology either implicitly assumes a remote attacker who can monitor or gather information about the processor's I/O (to DRAM) by executing his own malicious SW thread or assumes a physical attacker and explanations about how to counter physical attacks, see e.g. [14, 15]. In this paper we focus on the physical attacker (the reason why PUF research started in the first place) and we only wish to use Definition 2 to describe physical objects with a truly (in concrete terms) small TCB.

2.2 Strong PUFs vs Weak PUFs

By convention, if the challenge space C_λ is too large to be exhaustively enumerated by an adversary, we call PUF P_λ strong. Otherwise, it is called a weak PUF. The borderline between weak and strong is fuzzy in that the computational power of the adversary (which may change over time) decides what is weak and what is strong. An asymptotic definition for families of PUFs can be precise:

Definition 3. *A family of PUFs described by manufacturing processes $\{M_\lambda\}$ with challenge spaces $\{C_\lambda\}$ is called f-strong if at any moment of a PUF's execution there are $\Omega(f(\lambda))$ challenges in C_λ which do not lead to an empty response $\perp$, and is called weak if the number of these challenges is $O(1)$.*

Without giving a formal proof, we argue that since at least λ independent bits of information (in the form of manufacturing variations) for at least one unclonable response must be present within P_λ, the minimum possible size of P_λ is $O(\lambda)$ mm^3 (by the holographic bound by t'Hooft and Susskind a bit of information has a minimum physical size [16]). This observation which has been made in [17] as well, defined below, will help us in our analysis of PUFs:

Definition 4. *A family of stateless PUFs described by manufacturing processes $\{M_\lambda\}$ has bounded size if the physical size of the TCB (the assumed tamper-resistant part of) a physical object $P_\lambda \leftarrow M_\lambda$ is $O(\lambda)$ mm^3.*

If a family of physical objects can be proven to be a f-strong family of PUFs without assuming any computational hardness assumption, then R_λ must be represented by at least $O(f(\lambda) \cdot \lambda)$ bits, hence $f(\lambda) = O(1)$. This sketches the proof of the following lemma:

Lemma 1. *A family of PUFs described by manufacturing processes $\{M_\lambda\}$ with bounded size can only exist if (1) it is weak, or (2) its design is based on some computational hardness assumption.*

Note that a family of weak PUFs described by manufacturing processes $\{M_\lambda\}$ can be extended to simulate a family of 2^λ-strong PUFs by extending the PUF interfaces with a semantically secure variable length encryption algorithm Enc [5]. The new manufacturing process M'_λ uses M_λ to create a physical object with functionality $g(.) \leftarrow M_\lambda$ and extends the object with an additional interface which interprets a challenge of $\geq \lambda$ bits as a plain text and computes the response $r = Enc_{g(IV)}(c)$. Here, $K = g(IV)$, one of the weak PUF's responses, is used as secret key of the encryption.

In practice, however, additional public, i.e., not private, helper information needs to be written into the interface in order to correct for measurement noise of $g_\lambda(IV)$; the helper information differs from PUF to PUF, hence, the interface needs non-volatile storage. If this non-volatile storage is outside the TCB, then the adversary can tamper with the helper info with the aim to slightly change the secret key $K = g(IV)$, this results in a slightly different attacker's game with respect to the encryption scheme which needs to be included in the security assumption:

Lemma 2. *Given a semantically secure encryption scheme and a family of weak PUFs of bounded size, a family of 2^λ-strong stateful PUFs of bounded size can be constructed.*

The construction dramatically extends the TCB with a large-sized control circuitry which should not only (1) be tamper-resistant such that the logical functionality of the control circuitry cannot be maliciously changed, but should also (2) be private in that internal wire values of the control circuitry cannot be read by an adversary, since the wire values of the implemented encryption circuitry will have information about the key $K = g(IV)$.

For completeness, we mention that the existence of other kinds of families of 2^λ-strong PUFs with much smaller TCBs are into some extend in jeopardy as advanced attacking methods using e.g. Machine Learning (ML), which treat the PUF as a black box, have been able to break, i.e. software clone, existing strong PUF designs such as the arbiter PUF or the XOR of several arbiter PUFs [18, 19].

A recent breakthrough proposes the first stateless cryptographically secure PUF, whose security can be reduced to the hardness of Learning Parity with Noise (LPN) [20]. In particular, stateless implies that no wires need to be made disfunctional. As in this construction, a PUF interface (on top of a weak stateless PUF of bounded size) which simulates a 2^λ-strong PUF of bounded size is required to be tamper-resistant and private; its size is much smaller as it only needs to compute a Gaussian elimination and a hash function. [The main advantage of the new PUF is that it self-corrects all measurement noise without public helper information (which is needed in fuzzy extraction or error correction), and this means that no left-over hash lemma is needed to compress the response to a private string of bits; therefore, the LPN based PUF can correct much more measurement noise than what would otherwise be theoretically impossible (with helper information).]

Lemma 3. *Based on the hardness of LPN, a family of 2^λ-strong stateless PUFs of bounded size can be constructed from a family of weak stateless PUFs of bounded size.*

We notice that it still remains an open problem to design a strong PUF whose logical interface does not need to be private, i.e., even though tamper-resistance is required, is it possible to reduce the internal functionality that needs to remain private to only the PUF part which implements the functionality that depends on manufacturing variations?

Now that we have discussed the definitional landscape of PUFs and have formally introduced state in their interfaces, we will define and discuss erasable PUFs in the next section.

3 PLayPUFs: Programmable Logically Erasable PUFs

3.1 Erasable PUFs

To fix KE in the re-use model (or posterior access model), Rührmair et. al pointed out that a PUF should be strengthened with other complementary features, such as making the CPRs to be "erasable" [21]. To make this possible a PUF must have a form of non-volatile state to enable this erasure operation. In [22], erasability is defined by an extra interface function $ER(.)$ which represents a special erasure operation. If $ER(.)$ takes as input a challenge $\bar{c}$ of PUF P, it turns P into a physical system P' with the following properties:

1. P' has got the same set of possible challenges as P (and P' is again an erasable PUF). Let $\mathcal{E}$ be the set of previous inputs to $ER(.)$, including $\bar{c}$.
2. For all challenges $c \neq \bar{c}$, it holds that $g_{P'}(c) = g_P(c)$.
3. Given a list of all collected CRPs so far, $\bar{c}$, and black-box access to P', it is impossible to determine $g_P(\bar{c})$ with a probability that is substantially better than random guessing. Intuitively, the response $g_P(\bar{c})$ of the erasable PUF for challenge $\bar{c}$ has been erased in P' and for this reason we call the challenges in $\mathcal{E}$ erased. Notice that this property strengthens Definition 1 in that responses of erased challenges (besides those of unused challenges) can also not be cloned.

We adapt this definition to our framework: we interpret an input challenge as a pair consisting of the original input challenge and a flag indicating whether we want to erase the corresponding CRP or not.

Definition 5. [Erasable PUFs] *A family of strong PUFs with tamper-resistant state described by manufacturing processes $\{M_\lambda\}$ is called erasable if each algorithm $g \leftarrow M_\lambda$ has the property that it takes as input a challenge $(c, e) \in C_\lambda$ with $e \in \{0, 1\}$ together with state $s_{current} \in S_\lambda$ such that $g((c, e), s_{current}) = \bar{g}^{\mathcal{E}}((c, e), s_{current})$ where algorithm $\bar{g}$ is defined as follows:*

- *Initially we define the set of erased states $\mathcal{E} = \emptyset$.*
- *Whenever $(r, s_{new}) = g((c, e), s_{current})$ is executed, $\mathcal{E}$ is extended with c if and only if $e = 1$ (indicating that the corresponding response should be erased). If $e = 1$ or $c \in \mathcal{E}$, then $\bar{g}$ outputs $(\perp, s_{new})$, otherwise it outputs (r, s_{new}).*

Notice that $e = 1$ indicates to the PUF that the response corresponding to c should be erased. Algorithm $\bar{g}$ has oracle access to what has been erased in the past and outputs the empty response if the corresponding challenge has been erased before. By requiring $g = \bar{g}$ we know that the actual physical object implements this type of erasability as well. The unclonability property for erasable PUFs implies that if a response to a challenge has been erased ($e = 1$ in Definition 5), then an adversary without access to this response ($k = 0$ in Definition 2 for the interaction which generates the response) cannot replay back this previous state in which the PUF will issue the response again and is not able to gather information which can predict a sufficient number of response bits.

In our definition, we have tamper-resistant state $s_{current}$ which can be monitored by an adversary who is in possession of the PUF. If somehow state $s_{current}$ is physically integrated with the PUF functionality based on manufacturing variations, then one may believe in $s_{current}$ being private

as well. Up to now, it is still very difficult to implement this type of erasable PUF practically, because the PUF should be able to erase each CRP individually. This does not combine well in the popular PUF designs of [23,24]. Nevertheless, Rührmair et. al suggested to use a new SHIC (Super-High Information Content) PUFs as a candidate solution for erasable PUFs [22]. Because each pair of CRPs of a SHIC PUF is information theoretically independent, one is able to erase a CRP individually [25]. One limitation of SHIC PUFs is that they can only be read out at a slow rate (10^2 to 10^3 bits per second), while many of the other PUFs can be read out at 10^6 bits per second [22]. Hence, this solution does not fit a high throughput application; it also does not offer a general design principle applicable to all PUFs.

3.2 PLayPUFs: Programmable Logically Erasable PUFs

In this paper we propose to implement an erasable PUF by adding extra features not in the PUF design itself but in its logical interface as defined in Definition 5; this interface extension will implement the erasability functionality and can be used (as a general design principle) to turn any PUF into an erasable PUF. So, not the underlying PUF itself evolves as a dynamical physical system as proposed in [1,22], but the extended interface will evolve dynamically. Since the extended interface implements erasability logically, it needs non-volatile state in order to remember its dynamically changing state. We note that, in general, one of the advantages of using PUF technology is that it does not necessarily require any non-volatile memory to store digital state. In this paper, we sacrifice this advantage as it will allow us to gain other important benefits.

We can make the logically erasable PUF even stronger by programming a counter value *ctr* representing the number of times a response for a given challenge can be generated, i.e. once this number *ctr* is exceeded, challenge c ought to be automatically erased. This extra intelligence in the PUF interface allows forward as well as "backward" secure secret key management and an efficient key renewal without a master digital key, as explained in Section 4.

Definition 6. [PLayPUFs] *A family of strong PUFs with tamper-resistant state described by manufacturing processes $\{M_\lambda\}$ is called programmable logically erasable if each algorithm $g \leftarrow M_\lambda$ has the property that it takes as input a challenge $(c, ctr) \in C_\lambda$ with $ctr \in \{0, 1, \ldots\}$ together with state $s_{current} \in S_\lambda$ such that $g((c, ctr), s_{current}) = \bar{g}^E((c, ctr), s_{current})$ where algorithm $\bar{g}$ is defined as follows:*

- *Initially we define E as the function which maps each challenge in C_λ to ∞.*
- *Whenever $(r, s_{new}) = g((c, ctr), s_{current})$ is executed, if $E(c) \neq 0$, then $\bar{g}$ outputs (r, s_{new}) and $E(c)$ is updated to $min\{(E(c) - 1), ctr\}$ (indicating that the response corresponding to c moves at least one step closer to being erased), and if $E(c) = 0$ (the response corresponding to c has been erased), then $\bar{g}$ outputs $(\perp, s_{new})$.*

For completeness we notice that the logically reconfigurable PUF in [26] was the first to use tamper-resistant state for the purpose of reconfiguring *all* the CPRs together. Here, we provide a general framework of how to use tamper-resistant state in combination with PUFs which allows us to define PLayPUFs for erasure of *individual* CRPs, which in turn form the basic building block for constructing self-recovering CAs with backward security (see Section 4).

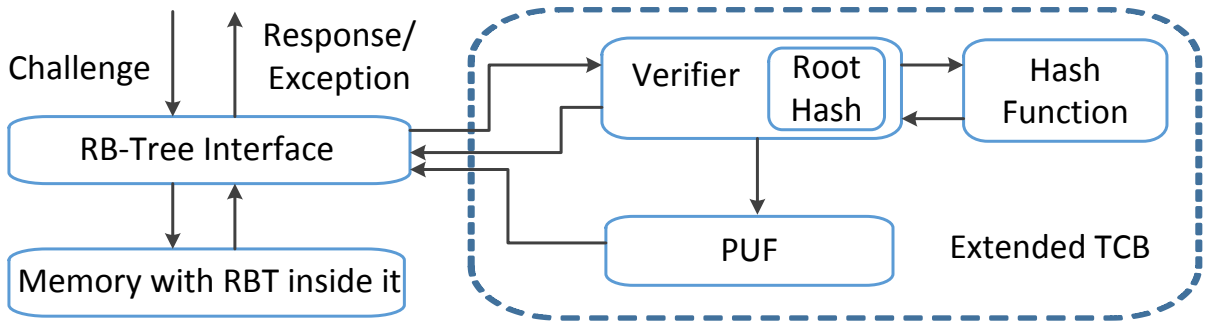


Figure 1: The entire system of Programmable Logically Erasable PUF

3.3 PLayPUF Design

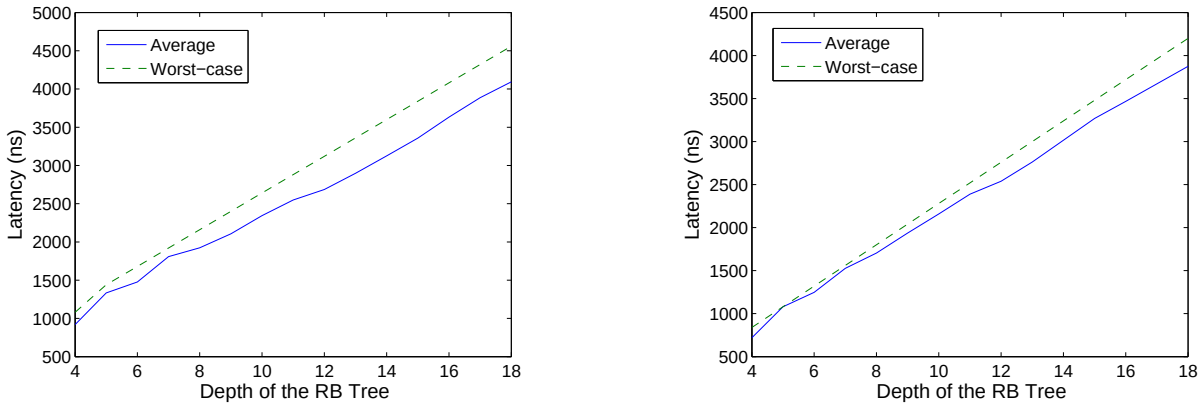
We propose to merge a Red-Black Tree [27, 28] and an Authenticated Search Tree [29] to construct a data structure which can be stored in public storage and which integrity and freshness can be verified using a small $O(\lambda)$ sized tamper-resistant state inside the TCB of the PUF. The tree structure records all challenges with their counter values.

For a new challenge c, the potentially untrusted tree structure must provide to the TCB (by using its authenticated search tree structure) a "proof of non-existence" for c so that the TCB allows a response for c to be computed. The tree needs to be updated with c and its counter value. Here, the rotation operation of the Red-Black Tree [27] structure of the tree is used to keep the tree balanced. This means that the depth of the tree will be proportional to the log of the number of nodes in the tree, for any access pattern.

For an already used challenge c as input, the tree structure must provide to the TCB (by using its authenticated search tree structure) a "proof of integrity and freshness" for c and its most recent counter value $E(c)$, see Definition 6. The TCB will check whether $E(c) \neq 0$ in which case it allows a response for c to be computed.

Fig. 1 shows our PLayPUF design. It consists of a public software interface with public memory/state and a hardware TCB which contains the PUF functionality based on manufacturing variations and a small $O(\lambda)$ sized tamper-resistant state in the form of the root-hash of the tree. Each node in the tree structure contains the value of a used challenge, the counter value associated with that challenge to indicate the number of times this challenge can still be used before being erased, three pointers pointing to its parent and its two children (if existing), and the color and hash value of this node; the hash is computed over the challenge, counter and the hash values stored in its children. The hashes are used to prove non-existence or integrity and freshness.

Intuitively, the PLayPUF works as follows: the software interface receives a challenge request and sends sufficient information from the untrusted memory to the TCB for verification. If the untrusted memory has been verified successfully and the requested challenge has not been erased, the TCB will decrement the counter, evaluate the PUF with that challenge, compute new hash values in the tree and update the trusted root hash value. With the new hash values computed by the TCB, the interface will update the untrusted memory accordingly. A detailed description and pseudocodes of our PLayPUF design are in Appendix B and Appendix C, respectively.



(a) Initialization (b) Extraction

Figure 2: The average and worst-case latency for serving one challenge request with respect to the depth of the RB Tree.

3.4 Evaluation

We have implemented our proposed architecture on Xilinx Kintex FPGA. Due to the fixed length of the inputs to the hash function, we decided to build a one-way function from AES-128 by the Davies-Meyer construction [30, 31]. We measured the performance of our implementation for challenge initialization (each challenge adds a new node in the tree) and response extraction (all challenges are existing in the tree) separately. Fig. 2 illustrates the average and worst-case latency of serving one request with respect to the depth of the RB-tree. We can see that the latency grows linearly with respect to the depth of the RB tree, which shows that the complexity of search, verification and update operations is only $O(lg(n))$, where n is the number of nodes in the tree. This performance is orders of magnitude higher than that of a SHIC PUF and comparable with the throughput of other strong PUFs [22].

4 Key Management

4.1 Forward and "Backward" Secure Key Management

Forward security is a widely-used technique to mitigate damage of exposure of secret keys [32]. If in a forward secure key update scheme a current secret key is stolen/leaked, then leakage of any previously used secret keys is prevented, implying that all previous encrypted data or communication will not be compromised.

Forward secure public key encryption was proposed in [3]. It allows secret keys to be updated at regular periods with forward security in that, if a cipher text y is the result of a public key encryption of x with public key PK and index i, then y can be decrypted using a secret key SK_i, and the leakage of SK_i does not expose $SK_1, SK_2, \ldots, SK_{i-1}$.

A complementary notion of "backward" security should guarantee that, if SK_i leaks, then also no information about $SK_{i+1}, SK_{i+2}, \ldots$ is exposed. Since the key update algorithm needs SK_i to compute next keys, leakage of SK_i would necessarily compromise all future keys. We notice that if we restrict ourselves to the keys represented by the leaves from the Binary Tree Encryption (BTE)

used in [3], then a key update solely based on such a key can only generate new children in the BTE and this can therefore never reach other leaves including those representing next keys. Also the properties of the used BTE are such that no useful information about a parent of such a leaf can be learned with which information about the keys of other leaves can be computed so that encryption based on those keys can be broken. Therefore, in the remainder of this paper we denote by $SK_1, SK_2, \ldots, SK_n$ the keys in the BTE which correspond to leaves.

Since the key update algorithm does not work anymore (which is necessary for backward security as explained above), all keys will need to be stored in secure memory. The secure memory may implement a trusted timer used for releasing keys from memory. To reduce the TCB of such a secure memory, a master key can be used to encrypt each of the SK_i. Now the whole key storage relies on keeping the master key private. Clearly, relying on a master key does not truly give a solution for backward security as its leakage will compromise the whole system. Is it possible to get away with a digital master key, can we somehow use a PUF as a sort of master key? We will show that with a PLayPUF we can do exactly this and even allow a stronger notion of backward security, see also Fig. 3:

Definition 7. [Backward Security] *(Sketch) A triple (Str, Int, Ext), representing an address to storage mapping Str, a stateful interface Int, and an extraction algorithm Ext, encodes a sequence of keys $\{SK_i\}$ if $SK_{i+1} = Ext^{Int,Str}(SK_i)$ for $i \geq 0$ with $SK_0 = \epsilon$, where oracle access to Int means that Int properly updates its own internal state.*

The triple (Str, Int, Ext) is called backward secure (for parameter λ) if the following property holds: If

1. *an adversary eavesdrops SK_j from the owner*
2. *after which the adversary with access to storage Str and black-box access to interface Int tampers with Str such that*
3. *if the owner reconstructs $SK_{i+1} = Ext^{Int,Str}(SK_i)$ for $j \leq i \leq h-1$ and $SK'_{h+1} = Ext^{Int,Str}(SK_h)$*
4. *where $h \geq j$ is the minimal index for which (a) $SK'_{h+1} \neq SK_{h+1}$ or (b) $SK'_{h+1} = SK_{h+1}$ is leaked to the adversary,*

then the owner can detect tampering or leakage of SK_{h+1} before he will ever use SK_{h+1} for the first time (with probability $\geq 1 - negl(\lambda)$).

Definition 7 implies that *if the owner has started using SK_i, an adversary cannot get any information about SK_i from storage Str through its interface at all.* To realize backward security, we propose to use a PLayPUF to securely encode and store the forward secure secret keys constructed as the leaves of the binary tree encryption in [3]. In an initialization phase, (a) all the forward secure secret keys SK_i are generated, (b) a random initial challenge c_1 is selected, (c) subsequent challenges are computed using

$$c_{i+1} = f(SK_i), \tag{2}$$

where $f(.)$ is a one-way function, (d) the PLayPUF is used to generate responses r_i with

$$(r_i, s_{i+1}) = \overline{g}^E((c_i, 1), s_i), \ s_1 = \epsilon \tag{3}$$

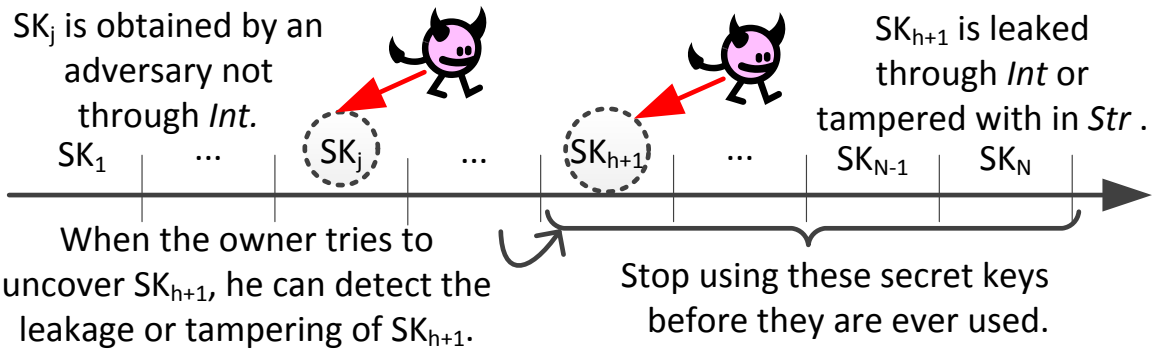


Figure 3: Backward security for secret key management according to Definition 7.

according to Definition 6, and (e) only the initial challenge c_1 and the masked keys

$$M_i = SK_i \oplus e(r_i) \text{ togther with } MAC_{SK_i}(M_i), \quad \text{for } i \geq 1, \tag{4}$$

where $e(.)$ uses privacy amplification [33] for extracting λ bits from r_i, are stored in the public storage Str.

Since each challenge c_i has its counter set to 1 during the initialization phase, each secret key SK_i can be retrieved exactly once after the initialization phase. In order to retrieve SK_{i+1} the legitimate owner, who knows the current key SK_i, simply asks the PLayPUF (which represents Int) for the response r_{i+1} corresponding to $c_{i+1} = f(SK_i)$ and computes $SK_{i+1} = M_{i+1} \oplus r_{i+1}$. If the PLayPUF responds with $\perp$ rather than r_{i+1} or if the MAC given the computed SK_{i+1} does not verify, then the legitimate owner concludes that SK_{i+1} has been compromised and will not start using SK_{i+1}. The legitimate owner only keeps the newly retrieved key SK_{i+1} and discards the old key SK_i.

Intuitively, if the legitimate owner properly retrieves SK_{i+1}, then he knows that the PLayPUF did not yet erase the corresponding CRP (c_{i+1}, r_{i+1}), hence, it has not been leaked through the PLayPUF to an adversary. In addition, once SK_{i+1} is retrieved the PLayPUF erases (c_{i+1}, r_{i+1}), hence. an adversary cannot retrieve information about SK_{i+1} in the future. The proof of the following theorem is sketched in Appendix A.

Theorem 1. *The above PLayPUF based key management scheme is backward secure.*

Our scheme realizes backward security by minimally extending a PUF's TCB with an $O(\lambda)$ tamper-resistant state. Backward security renders recent attacks and threats from insiders [34,35] as well as remote software attacks impossible. This is a crucial step forward in secure secret key management; with backward security a trusted/secure Public Key Infrastructure (PKI) is possible[2], without backward security (the current state of the art) PKI is believed to be under risk [36].

Our evaluation of our PLayPUF design shows that the proposed key management is very efficient: E.g., if secret keys are updated every 10 minutes for one year, we need 52560 challenge requests during initialization. This only takes $\approx 0.19s$ for our implementation on the Xilinx Kintex 7 FPGA.

[2]To be able to recover from failures or denial-of-service one may use a second properly isolated storage to backup all the keys with another PLayPUF.

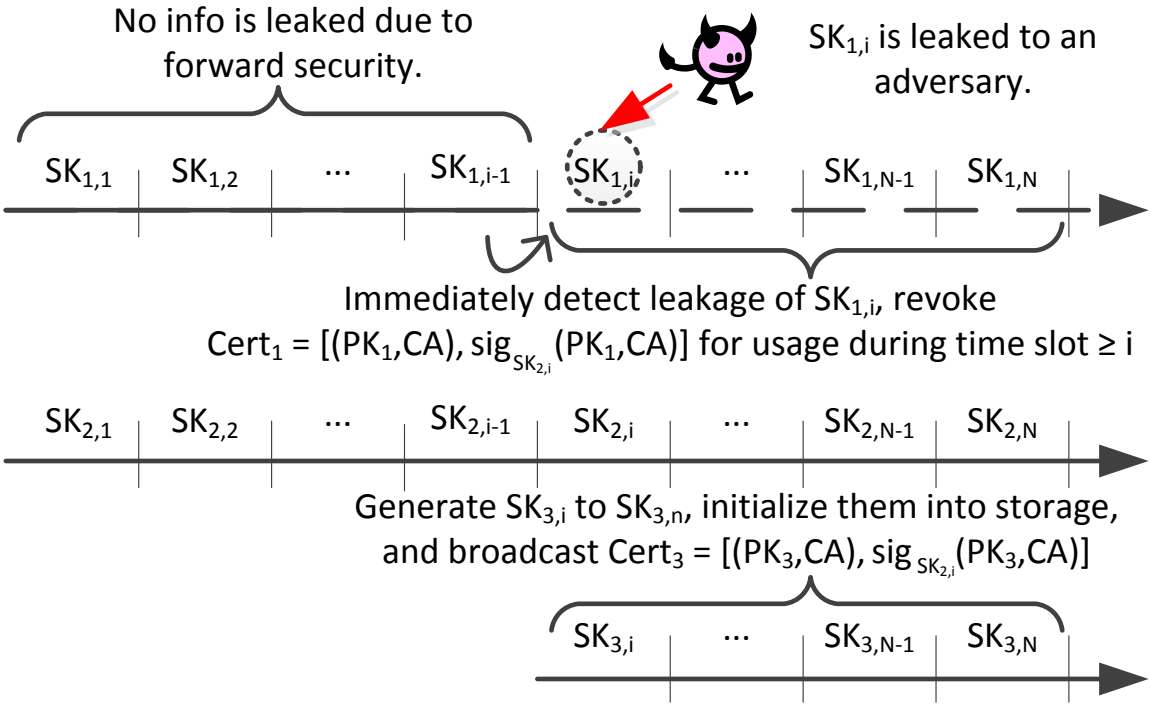


Figure 4: Self-Recovery Certificate Authority

4.2 Self-Recovering Certificate Authority

The above key management scheme can also be applied to forward secure signature schemes [37–43]. We propose to use the forward secure signature scheme in [38] which is constructed from a hierarchical identity based signature scheme and use a tree structure with the same property as explained above for the tree based encryption in [3], i.e., we can again use the keys at the leaves of the tree structure with each leaf not revealing any useful information about the other leaves. This allows us to construct a secure Certificate Authority (CA), which is very much needed given the recent key exposures and attacks originating from fake certificates [44].

The main concept is depicted in Fig. 4. The CA generates two public keys with corresponding forward and backward secure secret key sequences

$$(PK_1; SK_{1,1}, \ldots, SK_{1,N}) \text{ and } (PK_2; SK_{2,1}, \ldots, SK_{2,N})$$

for signing certificates. If an attacker obtains one of the secret keys $SK_{1,i}$ through the interface of the key storage, then the CA will immediately detect this leakage as soon as the CA attempts to reconstruct $SK_{1,i}$. After detection, the CA will start a self-recovery process: the CA (1) discards sequence $SK_{1,i}$ to $SK_{1,N}$, revokes the use of PK_1 for verification of certificates corresponding to time slots i, $i+1$, etc., but does not need to revoke PK_1 for verification of previously issued certificates as these are not fake, (2) generates a new public key with corresponding forward and backward secure secret key sequences $(PK_3; SK_{3,1}, \ldots, SK_{3,N})$, and (3) uses $SK_{2,i}$ (which is not leaked or tampered with) to certify PK_3. After this self-recovery process, the CA still maintains

two public keys with corresponding forward and backward secure secret key sequences (this concept can be generalized to more sequences if an adversary can get information about multiple keys within the same time slot).

Key Updates vs. One-Time Initialization. While existing forward secure encryption/signature schemes require efficient key updates, we only need an efficient one-time initialization of all keys in order to realize backward security but in such a way that, given a leaked key, learning information about other keys with which encryption/signatures can be broken/impersonated is hard. Our requirements seem weaker and we expect the possibility of more efficient forward and backward secure public key encryption/signature schemes.

5 Conclusion

We introduced the first formal definitional framework for PUFs with state. Within this framework we defined a new hardware security primitive, called a Programmable Logically Erasable PUF or PLayPUF for short. A PLayPUF can be constructed from any PUF by adding a stateful interface into the PUF's TCB. The extended TCB only assumes a small ($O(\lambda)$ where λ is a security parameter) tamper-resistant state (i.e., the state can be read by an adversary but cannot be modified without damaging the PLayPUF's functionality). Experiments show that our PLayPUF construction is efficient.

As a novel security guarantee we defined backward security for key management and we showed how this can be realized by using a PLayPUF. Backward security allows detection of leaked or tampered secret keys *before* they are being used. We showed how backward security (in combination with forward security) allows us to design a Certificate Authority (CA) who is able to recover from a leaked signing key in that a new public key and backward secure secret key sequence can be bootstrapped and (securely) certified by the CA itself. The old public key only needs to be invalidated for verifying new certificates but can still be used for verifying old certificates.

6 Acknowledgment

This project was supported in part by AFOSR MURI under award number FA9550-14-1-0351.

References

[1] U. Ruhrmair and M. van Dijk, "Pufs in security protocols: Attack models and security evaluations," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 286–300.

[2] M. van Dijk and U. Rührmair, "Physical unclonable functions in cryptographic protocols: Security proofs and impossibility results." *IACR Cryptology ePrint Archive*, vol. 2012, p. 228, 2012.

[3] R. Canetti, S. Halevi, and J. Katz, "A forward-secure public-key encryption scheme," in *Advances in Cryptology Eurocrypt 2003*. Springer, 2003, pp. 255–271.

[4] Y. Dodis, L. Reyzin, and A. Smith, "Fuzzy extractors: How to generate strong keys from biometrics and other noisy data," in *Advances in cryptology Eurocrypt 2004*. Springer, 2004, pp. 523–540.

[5] B. Gassend, M. V. Dijk, D. Clarke, E. Torlak, S. Devadas, and P. Tuyls, "Controlled physical random functions and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 10, no. 4, p. 3, 2008.

[6] S. Tajik, E. Dietz, S. Frohmann, J.-P. Seifert, D. Nedospasov, C. Helfmeier, C. Boit, and H. Dittrich, "Physical characterization of arbiter pufs," in *Cryptographic Hardware and Embedded Systems–CHES 2014*. Springer, 2014, pp. 493–509.

[7] C. Brzuska, M. Fischlin, H. Schröder, and S. Katzenbeisser, "Physically uncloneable functions in the universal composition framework," in *Advances in Cryptology CRYPTO 2011*. Springer, 2011, pp. 51–70.

[8] U. Rührmair, "Oblivious transfer based on physical unclonable functions," in *Trust and Trustworthy Computing*. Springer, 2010, pp. 430–440.

[9] R. Ostrovsky, A. Scafuro, I. Visconti, and A. Wadia, "Universally composable secure computation with (malicious) physically uncloneable functions," in *Advances in Cryptology–EUROCRYPT 2013*. Springer, 2013, pp. 702–718.

[10] D. Dachman-Soled, N. Fleischhacker, J. Katz, A. Lysyanskaya, and D. Schröder, "Feasibility and infeasibility of secure computation with malicious pufs," in *Advances in Cryptology CRYPTO 2014*. Springer, 2014, pp. 405–420.

[11] G. E. Suh, D. Clarke, B. Gassend, M. v. Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2003, p. 339.

[12] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: An extremely simple oblivious ram protocol," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 299–310.

[13] G. E. Suh, C. W. O'Donnell, and S. Devadas, "Aegis: A single-chip secure processor," *Information Security Technical Report*, vol. 10, no. 2, pp. 63–73, 2005.

[14] K. Tiri and I. Verbauwhede, "A logic level design methodology for a secure dpa resistant asic or fpga implementation," in *Proceedings of the conference on Design, automation and test in Europe-Volume 1*. IEEE Computer Society, 2004, p. 10246.

[15] S. Nikova, C. Rechberger, and V. Rijmen, "Threshold implementations against side-channel attacks and glitches," in *Information and Communications Security*. Springer, 2006, pp. 529–545.

[16] L. Susskind, "The world as a hologram," *Journal of Mathematical Physics*, vol. 36, no. 11, pp. 6377–6396, 1995.

[17] U. Rührmair, J. Sölter, and F. Sehnke, "On the foundations of physical unclonable functions." *IACR Cryptology ePrint Archive*, vol. 2009, p. 277, 2009.

[18] U. Ruhrmair, J. Solter, F. Sehnke, X. Xu, A. Mahmoud, V. Stoyanova, G. Dror, J. Schmidhuber, W. Burleson, and S. Devadas, "Puf modeling attacks on simulated and silicon data," *Information Forensics and Security, IEEE Transactions on*, vol. 8, no. 11, pp. 1876–1891, 2013.

[19] U. Rührmair, X. Xu, J. Sölter, A. Mahmoud, M. Majzoobi, F. Koushanfar, and W. Burleson, "Efficient power and timing side channels for physical unclonable functions," in *Cryptographic Hardware and Embedded Systems–CHES 2014*. Springer, 2014, pp. 476–492.

[20] C. Herder, L. Ren, M. van Dijk, M.-D. M. Yu, and S. Devadas, "A stateless cryptographically-secure physical unclonable function."

[21] M. van Dijk and U. Rührmair, "Protocol attacks on advanced puf protocols and countermeasures," in *Proceedings of the conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2014, p. 351.

[22] U. Rührmair, C. Jaeger, and M. Algasinger, "An attack on puf-based session key exchange and a hardware-based countermeasure: Erasable pufs," in *Financial Cryptography and Data Security*. Springer, 2012, pp. 190–204.

[23] B. Gassend, D. Lim, D. Clarke, M. Van Dijk, and S. Devadas, "Identification and authentication of integrated circuits," *Concurrency and Computation: Practice and Experience*, vol. 16, no. 11, pp. 1077–1098, 2004.

[24] J. W. Lee, D. Lim, B. Gassend, G. E. Suh, M. Van Dijk, and S. Devadas, "A technique to build a secret key in integrated circuits for identification and authentication applications," in *VLSI Circuits, 2004. Digest of Technical Papers. 2004 Symposium on*. IEEE, 2004, pp. 176–179.

[25] U. Rührmair, C. Jaeger, C. Hilgers, M. Algasinger, G. Csaba, and M. Stutzmann, "Security applications of diodes with unique current-voltage characteristics," in *Financial Cryptography and Data Security*. Springer, 2010, pp. 328–335.

[26] S. Katzenbeisser, Ü. Kocabaş, V. Van Der Leest, A.-R. Sadeghi, G.-J. Schrijen, and C. Wachsmann, "Recyclable pufs: Logically reconfigurable pufs," *Journal of Cryptographic Engineering*, vol. 1, no. 3, pp. 177–186, 2011.

[27] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein *et al.*, *Introduction to algorithms*. MIT press Cambridge, 2001, vol. 2.

[28] R. Bayer, "Symmetric binary b-trees: Data structure and maintenance algorithms," *Acta informatica*, vol. 1, no. 4, pp. 290–306, 1972.

[29] A. Buldas, P. Laud, and H. Lipmaa, "Accountable certificate management using undeniable attestations," in *Proceedings of the 7th ACM conference on Computer and communications security*. ACM, 2000, pp. 9–17.

[30] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC press, 1996.

[31] AES, NIST, "Advanced encryption standard," *Federal Information Processing Standard, FIPS-197*, vol. 12, 2001.

[32] M. Bellare and B. Yee, "Forward-security in private-key cryptography," in *Topics in Cryptology CT RSA 2003*. Springer, 2003, pp. 1–18.

[33] C. H. Bennett, G. Brassard, C. Crépeau, and U. M. Maurer, "Generalized privacy amplification," *Information Theory, IEEE Transactions on*, vol. 41, no. 6, pp. 1915–1923, 1995.

[34] M. Keeney, *Insider threat study: Computer system sabotage in critical infrastructure sectors*. US Secret Service and CERT Coordination Center, 2005.

[35] C. Colwill, "Human factors in information security: The insider threat–who can you trust these days?" *Information security technical report*, vol. 14, no. 4, pp. 186–196, 2009.

[36] C. Ellison and B. Schneier, "Ten risks of pki: What you're not being told about public key infrastructure," *Comput Secur J*, vol. 16, no. 1, pp. 1–7, 2000.

[37] M. Bellare and S. K. Miner, "A forward-secure digital signature scheme," in *Advances in Cryptology CRYPTO 99*. Springer, 1999, pp. 431–448.

[38] S. S. Chow, L. C. Hui, S. M. Yiu, and K. Chow, "Secure hierarchical identity based signature and its application," in *Information and Communications Security*. Springer, 2004, pp. 480–494.

[39] A. Kozlov and L. Reyzin, "Forward-secure signatures with fast key update," in *Security in communication Networks*. Springer, 2003, pp. 241–256.

[40] M. Abdalla and L. Reyzin, "A new forward-secure digital signature scheme," in *Advances in Cryptology ASIACRYPT 2000*. Springer, 2000, pp. 116–129.

[41] H. Krawczyk, "Simple forward-secure signatures from any signature scheme," in *Proceedings of the 7th ACM conference on Computer and communications security*. ACM, 2000, pp. 108–115.

[42] G. Itkis and L. Reyzin, "Forward-secure signatures with optimal signing and verifying," in *Advances in Cryptology Crypto 2001*. Springer, 2001, pp. 332–354.

[43] T. Malkin, D. Micciancio, and S. Miner, "Efficient generic forward-secure signatures with an unbounded number of time periods," in *Advances in Cryptology Eurocrypt 2002*. Springer, 2002, pp. 400–417.

[44] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer *et al.*, "The matter of heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*. ACM, 2014, pp. 475–488.

A Sketch Proof of Backward Security

Sketch Security Proof. Suppose the adversary acts as in Definition 7. We first consider case (b), i.e., SK_{h+1} is leaked or, equivalently, $e(r_{h+1})$ is leaked since $e(r_{h+1}) = M_{h+1} \oplus SK_{h+1}$ and M_{h+1} is stored in Str to which the adversary has access. Since $e(.)$ implements privacy amplification, if r_{h+1} contains λ bits of information unknown to the adversary then the complete string $e(r_{h+1})$ must be unknown to the adversary. For this reason an adversarial algorithm $\mathcal{A}$ must exist which, given knowledge of SK_j, Str, and black-box access to the PLayPUF (i.e., interface Int), teaches information about r_{h+1} in a way that $< \lambda$ bits entropy remain.

Notice that the recurrence relations (2), (3), and (4) create a functional sequence of dependencies $(r_{h+1}, .) = g(c_{h+1}, .)$, $c_{h+1} = f(SK_h)$, $SK_h = M_h \oplus e(r_h)$, $(r_h, .) = g(c_{h-1}, .)$, $c_{h-1} = f(SK_{h-1})$, $\ldots$, $c_{j+1} = f(SK_j)$. In this sequence the PLayPUF defined by $g(.,.)$ will produce the same (c_i, r_i) as computed during the initialization of storage Str (since the owner extracts the correct sequence $SK_{h+1}, \ldots, SK_j$ according to Definition 7). This means that algorithm $\mathcal{A}$ may as well be replaced by an algorithm $\mathcal{A}'$ which is like $\mathcal{A}$ but uses, instead of SK_j and Str, the value c_{h+1} which is closest related to r_{h+1} (we remind the reader that the SK_i are the leaves of the binary tree encryption in [3] and therefore do not have a sufficient algebraic relation that can be exploited). Since algorithm $\mathcal{A}'$ with only knowledge of c_{h+1} and black-box access to the PLayPUF leaks r_{h+1} in a way that $< \lambda$ bits entropy remain, Definition 2 implies that, for some time t, the adversarial algorithm $\mathcal{A}'$ (and therefore also algorithm $\mathcal{A}$) must have accessed the PLayPUF with challenge c_{h+1} and received back the response r_{h+1}. As a consequence, see Definition 6 and Equation (3), (1) r_{h+1} will be erased by the PLayPUF after time t and (2) before time t the legitimate owner did not yet challenge the PLayPUF with c_{h+1} (otherwise, $\mathcal{A}$ did not receive r_{h+1} but received $\perp$ instead and no information about r_{h+1} could have been leaked). This means that the owner tries to reconstruct SK_{h+1} after time t and will receive $\perp$ from the PLayPUF. This means that the owner detects the leakage of SK_{h+1} before ever using SK_{h+1}.

We now consider case (a), i.e., SK_{h+1} is tampered with and modified into $SK'_{h+1} \neq SK_{h+1}$. If SK_{h+1} is leaked or if the adversary gained information about r_{h+1} such that $< \lambda$ bits entropy remain, then he will be detected as explained above in case (b). So, if the adversary will not be detected, then SK_{h+1} is not leaked and the adversary does not get information about r_{h+1} such that $< \lambda$ bits entropy remain. This implies that the adversary cannot distinguish $e(r_{h+1})$ from a random λ-bit string, which in turn implies that the adversary cannot distinguish $M_{h+1} = SK_{h+1} \oplus e(r_{h+1})$ from a random λ-bit string.

The owner reconstructs from tampered memory M'_{h+1} the modified secret key as $SK'_{h+1} = M'_{h+1} \oplus e(r_{h+1})$ (by Definition 7, the owner properly extracts SK_h; given SK_h the owner properly extracts r_{h+1}, see Equations (2,3). Since $SK'_{h+1} \neq SK_{h+1}$, $M_{h+1} = SK_{h+1} \oplus e(r_{h+1}) \neq SK'_{h+1} \oplus e(r_{h+1}) = M'_{h+1}$. So, in order to pass undetected the adversary will need to, without knowledge of SK_{h+1} but with knowledge of $MAC_{SK_{h+1}}(M_{h+1})$ for (in his view) a random M_{h+1}, construct $MAC_{SK_{h+1}}(M'_{h+1})$ for some $M'_{h+1} \neq M_{h+1}$. This would break the security of the MAC.

We conclude that the PLayPUF based key management scheme is backward secure.

B The Interface of a PLayPUF

Part of a PLayPUF is an interface which consists of a public part and a TCB part as depicted in Fig. 5. We remind the reader that the TCB part consist of a tamper-resistant and private PUF

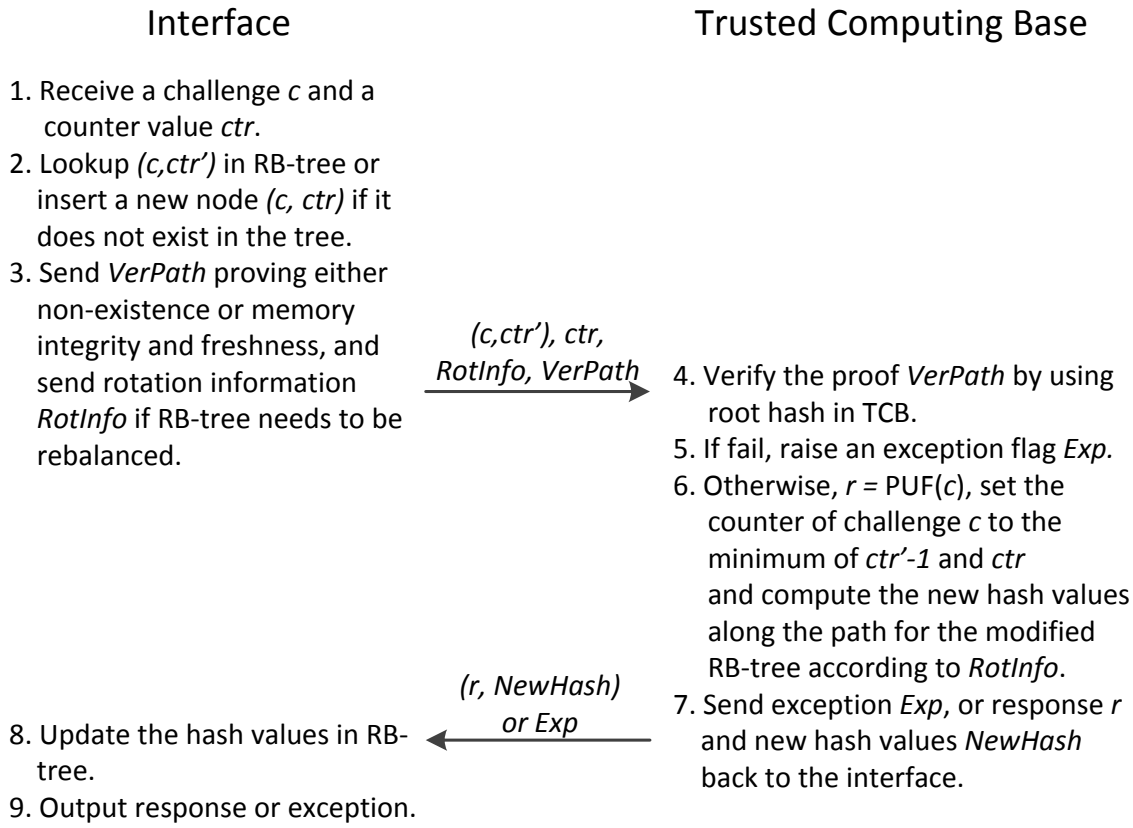


Figure 5: The protocol between software information and hardware TCB of a Programmable Logically Erasable PUF.

together with tamper-resistant additional circuitry which includes non-volatile state, see Fig. 1.

When a PLayPUF receives from a user (1) a challenge c with counter ctr as input, the PLayPUF will first use the public part of the interface circuitry to (2) lookup challenge c in the RB tree. The authenticated tree structure of the RB tree allows the public part of the interface to either compute a proof of non-existence of c (if c does not exists in the tree) or a proof of integrity and freshness of the retrieved c with its current counter value ctr' (if c already exists in the tree). A proof $VerPath$ consists of the hashes of the siblings of the path from c (if it exists) or from the leaf at which the new node is inserted (if c does not exist) to the root of the RB tree together with the values of the nodes on the path. When such a proof is (3) transmitted and (4) received by the TCB part of the interface, then the TCB part of the interface is able to hash all this information together in order to reconstruct the root of the tree, which it can then verify against its own copy in its tamper-resistant non-volatile state. If (5) verification fails or $ctr' = 0$ (in case c already exists in the tree), then either the public memory of the public interface was corrupted or by our definition of programmable logical erasability c must be considered erased; in either case an exception flag $\perp$ is returned.

If c did not exist, then besides a proof of non-existence also (3) rotation information is transmitted for inserting a new node (c, ctr). This rotation information is needed for maintaining the red-black invariant of the RB tree such that its balance remains guaranteed. It contains how many

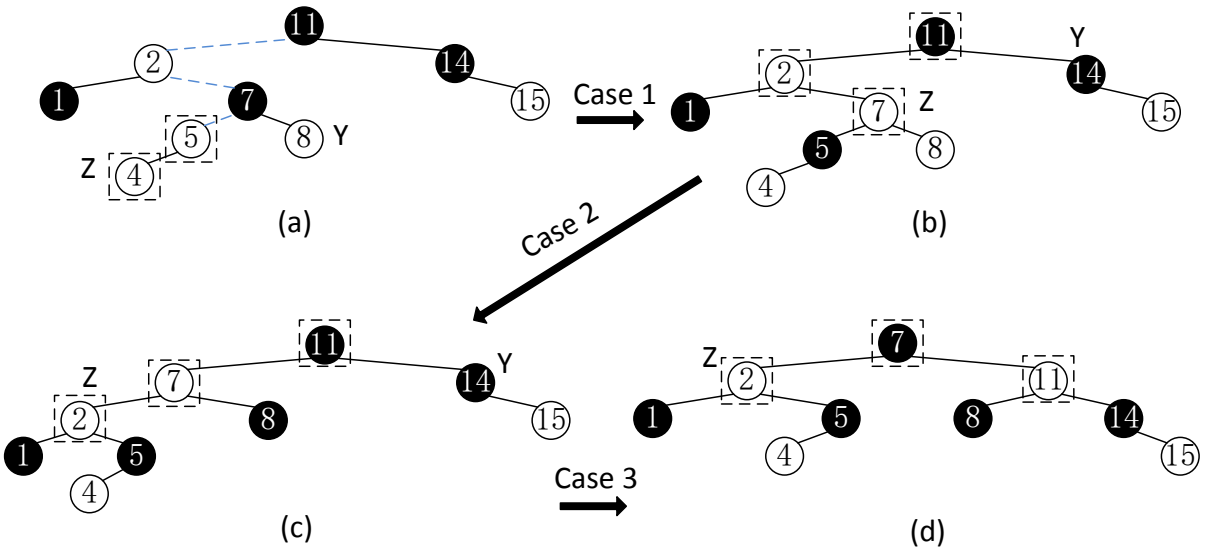


Figure 6: Insertion of a new node 4.

tree rotations happened (it cannot be more than two for one insertion operation), the direction of each rotation and the position of the tree rotation, below we explain a detailed example in Fig. 6. The rotation information can be computed by the public part of the interface as the complete RB tree is stored in its public memory; the public interface is in charge of maintaining the balance. Of course an adversary may try to corrupt this, but this will not have any consequences for the security/unclonability of the PLayPUF; the red-black structure is only added for improved performance in case of worst-case access patterns. The TCB part does not have access to the whole RB tree and is therefore informed by the public part of the interface how to recompute the root hash such that it corresponds to the newly balanced/rotated tree. Besides the rotation information itself, the public part if the interface also needs to transmit a proof of integrity and freshness of the couple of nodes on which the rotation depends (which turns out to already be present in $VeriPath$). All of this is contained in $RotInfo$ which is (3) sent to the TCB.

If all verifies correctly in (4-5), then the TCB will (6) set c's counter value to the minimum of ctr and $ctr' - 1$ (where $ctr' = \infty$ if c did not already exist), update the root hash in TCB by using the rotation information and/or new value of the node representing c, evaluate the PUF with this c, and (7) reply to the public interface the response r from the PUF. In our design we also let the TCB (7) transmit the updated hash information $NewHash$ to the public part of the interface (which could also have computed this itself). Next, the public interface will (8) update the RB tree in its public memory accordingly. and (9) output the response r of the PUF to the user or simply raise an exception flag.

Example Rotation. Fig. 6 depicts an example of consecutive operations in Red-Black Tree Insert-Fixup, see [27]. (a) A new node 4 is inserted. The dashed path in (a) is $VerPath$. All of the information in nodes 5, 7, 2 and 11 are included in $VerPath$, together with the hash values of nodes 8, 1 and 14, called the sibling's hash values. In order to verify non-existence, we need to reconstruct the root hash using $VerPath$ and compare with the trusted root hash stored in the TCB. In addition, we need to check whether new node 4 is added at the correct location, which

means $2 < 4 < 5$, and node 5 has no left child. Here, case 1 in [27] applies, so node 5 and 7 are recolored but the structure remain the same. There are six possible cases in a RB tree fixup, in which only case 2, 3, 5 and 6 in [27] will rotate the structure of the tree; this example shows three cases (the other three cases are similar in that they are mirrored versions of the three in the example). In (b),(c) and (d), the nodes in dashed blocks are the nodes which hash values need to be updated; the transition from (b) to (c) is a rotation and the transition from (c) to (d) is a rotation. Note that, $VerPath$ already provides all the information needed for updating these hash values. In this example, in order to compute the hash of node 2, 7 and 11 in (d), we need the hash value of node 5, which was updated in case 1 during the transition from (a) to (b), and the hash values of nodes 1, 8 and 14, which are exactly the sibling's hash values that are contained in $VerPath$.

C Pseudocodes of PLayPUF Implementation

Algorithm 1 RB-Tree-Interface. (Note that this code is modified based on **RB-Insert** in [27]. The lines added by us are indicated by *.)

```
1: procedure RB-Tree-Interface(Challenge C, Counter ctr, Red-Black Tree T)
2:     x = T.root
3:     y = T.nil
4:     z = T.nil                                                              ▷ *
5:     z.key.ch = C                                                           ▷ *
6:     i = 0                                                                  ▷ *
7:     while x ≠ T.nil do
8:         y = x
9:         if z.key.ch < x.key.ch then
10:            x = x.left
11:        else if z.key.ch > x.key.ch then
12:            x = x.right
13:        else
14:            Query.ch, Query.ctr, Query.lhash, Query.rhash = (x.key.ch, x.key.ctr, x.left.key.hash,
   x.right.key.hash)                                                          ▷ *
15:            Return Query, ctr, NULL, i, VerPath                            ▷ *
16:        end if
17:        VerPath[i].ch, VerPath[i].ctr, VerPath[i].shash = (x.p.key.ch, x.p.key.ctr, x.sibling.hash)
   ▷ *
18:        i = i + 1                                                          ▷ *
19:    end while
20:    VerPath[i].ch, VerPath[i].ctr, VerPath[i].shash = (x.p.key.ch, x.p.key.ctr, x.sibling.hash) ▷ *
21:    z.p = y
22:    if y == T.nil then
23:        T.root = z
24:    else if z.key.ch < y.key.ch then
25:        y.left = z
26:    else
27:        y.right = z
28:    end if
29:    z.left = T.nil
30:    z.right = T.nil
31:    z.color = RED
32:    RotInfo ← RB-Insert-Fixup(T, z)
33:    z.key.ctr = ∞                                                          ▷ *
34:    Query.ch, Query.ctr, Query.lhash, Query.rhash = (z.key.ch, z.key.ctr, z.left.key.hash,
   z.right.key.hash)                                                          ▷ *
35:    Return Query, ctr, RotInfo, i, VerPath                                 ▷ *
36: end procedure
```

Algorithm 2 RB-Insert-Fixup. (Note that the code is modified based on **RB-Insert-Fixup** in [27]. The lines added by us are indicated by *. Also, the pseudocodes of **Left-Rotate** and **Right-Rotate** can be found in [27]).

```
1: procedure RB-Insert-Fixup(Red-Black Tree T, Newly Inserted Node z)
2:     Initialize RotInfo.case1, RotInfo.case2, RotInfo.case3, RotInfor.case4, RotInfo.case5, RotInfo.case6 to 0 ▷ *
3:     while z.p.color == RED do
4:         if z.p == z.p.p.left then
5:             y = z.p.p.right
6:             if y.color == RED then ▷ Case 1
7:                 z.p.color = BLACK
8:                 y.color = BLACK
9:                 z.p.p.color = RED
10:                z = z.p.p
11:                RotInfo.case1 = RotInfo.case1 + 2 ▷ *
12:            else
13:                if z == z.p.right then ▷ Case 2
14:                    z = z.p
15:                    LEFT-Rotate(T, z)
16:                    RotInfo.case2 = 1 ▷ *
17:                end if
18:                z.p.color = BLACK ▷ Case 3
19:                z.p.p.color = RED
20:                Right-Rotate(T, z.p.p)
21:                RotInfo.case3 = 1 ▷ *
22:            end if
23:        else
24:            (same as then clause with "right" and "left" exchanged, and "Case 1, 2, 3" replaced with "Case 4, 5, 6")
25:        end if
26:    end while
27:    T.root.color = BLACK
28:    Return RotInfo ▷ *
29: end procedure
```

Algorithm 3 TCB

```
1: procedure TCB(Query Query, Counter Value ctr, Rotation Information RotInfo, Length of
   Proof N, Proof VerPath, Trusted Root Hash root)
2:     if Query.ctr == 0 then
3:         Exp = 1
4:         Return Exp, NULL, NULL
5:     else
6:         Exp ← VERIFY-PROOF(N, VerPath, Query, root)
7:         if Exp == 1 then                                          ▷ Verification failed
8:             Return Exp, NULL, NULL
9:         else                                                      ▷ Passed verification
10:            R ← PUF(Query.ch)
11:            NewHash, root ← UPDATE-HASH(N, VerPath, Query, ctr, RotInfo)
12:            Return Exp, R, NewHash
13:        end if
14:    end if
15: end procedure
```

Algorithm 4 Verify-Proof

```
1: procedure VERIFY-PROOF(Length of Proof N, Proof VerPath, Query Query, Trusted Root
   Hash root)
2:     if Query.ctr == ∞ then                                        ▷ Newly added node
3:         h = 0
4:     else                                                          ▷ Existing node
5:         h = Hash(Query.ch || Query.ctr || Query.lhash || Query.rhash)
6:     end if
7:     ch = Query.ch
8:     for i ← N − 1, 0 do
9:         if ch < VerPath[i].ch then
10:            h = Hash(VerPath[i].ch || VerPath[i].ctr || h || VerPath[i].shash)
11:        else
12:            h = Hash(VerPath[i].ch || VerPath[i].ctr || VerPath[i].shash || h)
13:        end if
14:        ch = VerPath[i].ch
15:    end for
16:    if h == root then
17:        Exp = 0
18:    else
19:        Exp = 1
20:    end if
21:    Return Exp
22: end procedure
```

Algorithm 5 Update-Hash

1: **procedure** UPDATE-HASH(Length of Proof N, Proof *VerPath*, Query *Query*, Counter Value *ctr*, Rotation-Information *RotInfo*)
2: **if** $Query.ctr = \infty$ **then** ▷ Newly added node
3: $j = N - 1$
4: **else** ▷ Existing node
5: $j = N - 2$
6: **end if**
7: $Query.ctr = min(Query.ctr - 1, ctr)$
8: $i = 0$
9: $NewHash[i++] = Hash(Query.ch \,||\, Query.ctr \,||\, Query.lhash \,||\, Query.rhash)$
10: $ch = Query.ch$
11: **if** $RotInfo \neq NULL$ **then**
12: **while** $i < (RotInfo.case1 + RotInfo.case4)$ **do** ▷ Case 1 and 4
13: $(ch, NewHash, i, j) \leftarrow$ HASH-NO-ROTATION$(ch, VerPath, NewHash, i, j)$
14: **end while**
15: **if** $RotInfo.case2 == 1$ **then** ▷ Case 2
16: $NewHash[i] = Hash(VerPath[j-1].ch \,||\, VerPath[j-1].ctr \,||\, VerPath[j-1].shash \,||\, NewHash[i-1])$
17: $NewHash[i+1] = Hash(VerPath[j-2].ch \,||\, VerPath[j-2].ctr \,||\, VerPath[j].shash \,||\, VerPath[j-2].shash)$
18: $ch, NewHash[i+2] = VerPath[j].ch, Hash(VerPath[j].ch \,||\, VerPath[j].ctr \,||\, NewHash[i] \,||\, NewHash[i+1])$
19: **else if** $RotInfo.case3 == 1$ **then** ▷ Case 3
20: $NewHash[i] = Hash(VerPath[j].ch \,||\, VerPath[j].ctr \,||\, VerPath[j].shash \,||\, NewHash[i-1]$
21: $NewHash[i+1] = Hash(VerPath[j-2].ch \,||\, VerPath[j-2].ctr \,||\, VerPath[j-1].shash \,||\, VerPath[j-2].shash)$
22: $ch, NewHash[i+2] = VerPath[j-1].ch, Hash(VerPath[j-1].ch \,||\, VerPath[j-1].ctr \,||\, NewHash[i-2] \,||\, NewHash[i-1])$
23: **else**
24: (same as **then** clauses for case 2 and 3 with the order of two children's hash values exchanged)
25: $i = i + 3$
26: $j = j - 3$
27: **end if**
28: **end if**
29: **while** $j \geq 0$ **do** ▷ No Fixup
30: $(ch, NewHash, i, j) \leftarrow$ HASH-NO-ROTATION$(ch, VerPath, NewHash, i, j)$
31: **end while**
32: $root = NewHash[i-1]$
33: Return *NewHash, root*
34: **end procedure**

Algorithm 6 Hash-No-Rotation

procedure HASH-NO-ROTATION(Challenge ch, Proof $VerPath$, NewHash $NewHash$, Index for NewHash i, Index for Proof j)
  if $ch < VerPath[j]$ **then**
    $ch, NewHash[i++] = VerPath[j].ch, Hash(VerPath[j].ch \;||\; VerPath[j].ctr \;||\; NewHash[i-1] \;||\; VerPath[j].shash)$
  else
    $ch, NewHash[i++] = VerPath[j].ch, Hash(VerPath[j].ch \;||\; VerPath[j].ctr \;||\; VerPath[j].shash \;||\; NewHash[i-1])$
  end if
  $j--$
  Return $ch, NewHash, i, j$
end procedure

Algorithm 7 Update-Tree

procedure UPDATE-TREE(Rotation Information $RotInfo$, New Hash $NewHash$, Newly Inserted Node z, Counter Value ctr, Red-Black Tree T)
  $z.key.hash = NewHash[0]$
  $z.key.ctr = min(ctr, z.key.ctr - 1)$
  $x = z.p$
  $i = 1$
  if $RotInfo \neq NULL$ **then**
    for $i \leftarrow 1, (RotInfo.case1 + RotInfo.case4)$ **do** ▷ Case 1 and 4
      $x.key.hash = NewHash[i]$
      $x = x.p$
    end for
    if $RotInfo.case3 == 1 or RotInfo.case6 == 1$ **then** ▷ Case 2,3,5,6
      $x.key.hash = NewHash[i]$
      $x.sibling.key.hash = NewHash[i+1]$
      $x.p.key.hash = NewHash[i+2]$
      $x = x.p.p$
      $i = i + 3$
    end if
  end if
  while $x \neq T.root$ **do**
    $x.key.hash = NewHash[i++]$
    $x = x.p$
  end while
end procedure