Efficient Erasable PUFs from Programmable Logic and Memristors

Yansong Gao[1], Chenglu Jin[2*], Jeeson Kim[3], Hussein Nili[4], Xiaolin Xu[5], Wayne Burleson[6], Omid Kavehei[7], Marten van Dijk[2], Damith C. Ranasinghe[8] and Ulrich Rührmair[9]

[1] Nanjing University of Science and Technology yansong.gao@njust.edu.cn
[2] University of Connecticut, {chenglu.jin,marten.van_dijk}@uconn.edu
[3] Nano-Neuro-Inspired Research Laboratory, RMIT University
[4] University of California, Santa Barbara
[5] University of Florida xiaolinxu@ece.ufl.edu
[6] University of Massachusetts, Amherst burleson@ecs.umass.edu
[7] The University of Sydney
[8] University of Adelaide, damith.ranasinghe@adelaide.edu.au
[9] Horst Görtz Institute for IT-Security, Ruhr Universität Bochum ruehrmair@ilo.de

Abstract.
At Oakland 2013, Rührmair and van Dijk showed that many advanced PUF (Physical Unclonable Function)-based security protocols (e.g. key agreement, oblivious transfer, and bit commitment) can be vulnerable if adversaries get access to the PUF and reuse the responses used in the protocol after the protocol execution. This observation implies the necessity of erasable PUFs for realizing secure PUF-based protocols in practice. Erasable PUFs are PUFs where the responses of any single challenge-response pair (CRP) can be selectively and dedicatedly erased, without affecting any other responses.
In this paper, we introduce two practical implementations of erasable PUFs: Firstly, we propose a full-fledged logical version of an erasable PUF, called programmable logically erasable PUF or PLayPUF, where an additional constant-size trusted computing base keeps track of the usage of every single CRP. Knowing the query history of each CRP, a PLayPUF interface can *automatically* erase an individual CRP, if it has been used for a certain number of times. This threshold can be programmed a-priori to limit the usage of a given challenge in the future before erasure.
Secondly, we introduce two nanotechnological, memristor-based solutions: mrSHIC-PUFs and erasable mrSPUFs. The mrSHIC-PUF is a weak PUF in terms of the size of CRP space, and therefore its readout speed has to be limited intentionally to prolong the time for exhaustive reading. However, each individual response can be *physically* altered and erased for good. The erasable mrSPUF, as the second proposed physical erasable PUF, is a strong PUF in terms of the size of CRP space, such that no limit on readout speed is needed, but it can only erase/alter CRPs in groups. Both of these two physical erasable PUFs improve over the state-of-the-art erasable SHIC PUF, which does not offer reconfigurability of erased CRPs making the erasable SHIC PUF less practical.

*Yansong Gao and Chenglu Jin are in alphabetical order and share first authorship. Yansong Gao contributed to Section 4 under the supervision of Damith C. Ranasinghe. Chenglu Jin contributed to Section 2 and 3 under the supervision of Marten van Dijk. Jeeson Kim, Hussein Nili and Omid Kavehei were in charge of fabricating the memristor based erasable PUFs. Xiaolin Xu and Wayne Burleson helped with implementing the PUF circuitry for the programmable logically erasable PUF. Ulrich Rührmair conceived and led the whole project.

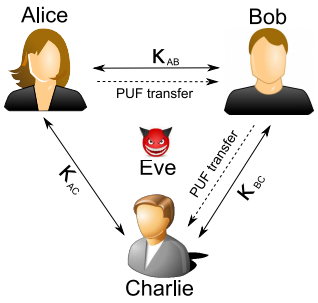


Figure 1: PUF based session key exchange involved three parties.

In passing, we contextualize and locate our new PUF type in the existing landscape, illustrating their essential advantages over variants like reconfigurable PUFs.

Keywords: Physical Unclonable Functions · PUF Re-use Model · Memristor · Erasable PUFs · Reconfigurable PUFs · PLayPUFs · mrSHIC-PUFs · mrSPUFs

1 Introduction and Overview

Since physical unclonable functions have been introduced as a security primitive [GCvDD02b, PRTG02], a variety of applications have been proposed [LLG+05, SD07, LLG+04], including many advanced cryptographic protocols, e.g. Key Agreement, Oblivious Transfer and Bit Commitment [BFSK11, Rüh10, OSVW13].

To show security vulnerabilities in the so-called "*PUF re-use model*" as proposed in [RvD13], we pick a simple PUF-based key exchange protocol as an example for our dicussion below. Figure 1 illustrates how a secure PUF-based key exchange protocol can be compromised by PUF re-use model in practice. Following the PUF re-use model introduced in [RvD13], an adversary is allowed to access the PUF used in the protocol and apply arbitrary challenges to obtain corresponding responses. The adversary can also eavesdrop on the communication channel.

Alice issues the PUF, evaluates two sets of CRPs (Challenge response pairs), and then delivers the PUF to Bob. To establish a session key, κ_{AB}, with Bob, Alice sends the first challenge set to Bob. Bob applies the received challenge set and obtains the responses that are treated as the session key, κ_{AB}. Bob then evaluates a new set of CRPs and sends the PUF to Charlie. Bob can now establish a session key, κ_{BC}, with Charlie using the new set of CRPs Bob saved. While Alice can also establish a session key, κ_{AC}, with Charlie using her second set of CRPs. Eve is able to eavesdrop on the first set of challenges that Alice sends to Bob and then apply the eavesdropped challenges to obtain the corresponding responses when Eve can physically access the PUF during the PUF transfer from Bob to Charlie. This implies that κ_{AB} has been compromised by Eve under PUF re-use model.

To solve this problem, Rührmair and van Dijk suggested the usage of erasable PUFs [RvD13]. The concept of erasable PUFs was introduced in [RJA11], meaning that users are allowed to irreversibly and selectively erase/alter the responses of a challenge without affecting the other CRPs. If an erasable PUF is used in the above key exchange protocol, Bob should erase the challenge set for deriving κ_{AB}, and thus Eve is not able to read out those CRPs. This secures the PUF-based key exchange protocol. Notice that, another PUF variant called reconfigurable PUF [KKvDL+11] can not help in this case, since reconfigurable PUFs erase the entire challenge space, so all the CRPs will be altered after reconfiguration. This can prevent the attacks in PUF re-use model, but it may affect the further usage of this PUF, e.g. in the scanerio of Figure 1, Alice and Charlie cannot establish secret key without another physical transfer of PUF. For the analysis on

more PUF-based cryptographic protocols under PUF re-use model, readers are referred to [RvD13] for a detailed discussion.

Considering the importance of this issue, its solution seems long overdue as only one erasable PUF has been introduced so far [RJA11]. In this paper, we present two ways of implementating erasable PUFs: Programmable Logically Erasable PUFs and Memristor based Physical Erasable PUFs. Both of them have unique advantages comparing with the existing erasable PUF design [RJA11].

Programmable logically erasable PUFs (PLayPUFs) realize the erasability by adding a digital interface to keep track of the history of applied challenges. By merging an authenticated search tree [BLL00] and a red-black tree [CLR+01, Bay72], we only need a constant size non-volatile memory (NVM) and hash computation to be added into the trusted computing base (TCB) to verify an arbitrarily large query history. We assume that adversaries are not able to tamper with the computation and the NVM in the trusted computing base, and the memory content of NVM is public. All the computation and memory outside of TCB, namely the untrusted part, can be maliciously tampered with and read out, and our system can still stay secure. The methodology of PLayPUF is generic, because it can be added to any secure PUF designs to introduce erasability feature. An erasure in PLayPUF is realized by denying the access to erased CRPs by its interface. Notice that, because the CRP space of a strong PUF is exponentially large in size, it is acceptable for users to permanently erase some CRPs of a strong PUF, but permanent erasure of CRPs in weak PUFs will deplete the CRPs very quickly, so we do not recommend to build a PLayPUF based on a weak PUF.

As the only existing erasable PUF design so far, erasable SHIC PUF is built on top of a crossbar structure, and its erasure operation requires the destruction of an underlying physical structure (a diode), such that the CRP created by this structure is inaccessible/erased forever [RJA11]. This one-time erasure limitation depletes the entire CRP space very fast, and SHIC PUF only has polynomial number of CRPs, so this one-time erasure limitation also limits its usage as an erasable PUF. Another drawback of erasable SHIC PUF is that since it only has a polynomial size CRP space, it has to intentionally limit the readout speed and therefore throughput to increase the difficulty for exhaustive readout.

Memristor, as an emerging nano-technology, has a unique physical phenomenon called cycle-to-cycle variation, which means that every time a memristor cell is reprogrammed, the user is not able to precisely control its resistance [YSS13, WS15, GRAS+16]. Leveraging this phenomenon, we are able to build two types of memristor based erasable PUF (mrSHIC-PUF and erasable mrSPUF), for which the erasure operation simply means reprogramming a certain cell or a certain group of cells. By introducing the reconfigurability of individual CRPs, mrSHIC PUF is able to reuse the underlying physical structure after an erasure operation. This implies that the CRP space, will not be depleted, even though it is also polynomial size. Of course, to build a secure erasable PUF, the new response of an erased challenge should be random and independent of its previous response. However, mrSHIC-PUF inherits the size of CRP space from SHIC PUF, so mrSHIC-PUF also needs to have a throttling mechanism to limit its readout speed, so that an adversary cannot enumerate all CRPs exhaustively.

The erasable mrSPUF, as the second proposed physical erasable PUF design, is a strong PUF, according to the size of its CRP space. Therefore, no more limited readout speed is needed, but it can only erase/alter the responses in a more coarse-grained way, meaning that it has to erase/alter CRPs in a small number of possible erasable sets, instead of erasing CRPs individually. This reduces the flexibility of the erasure operation, but it will still be of interest for PUF-based cryptographic protocols. Again, in order not to deplete the CRP space (notice there are only a polynomial number of possible erasable sets that partitions the whole exponentially large CRP space), we need reconfigurability of each

group of CRPs.

1.1 Our Contributions

We made three significant contributions in this paper:

1. We provide a unified formal definitional framework of PUF, which captures the properties of strong PUFs, weak PUFs, stateless PUFs, stateful PUFs and erasable PUFs.

2. We introduce programmable logically erasable PUFs, which can be added as an interface to any PUFs to realize erasability. As an additional property, we gain programmability for free, which means that users can precisely define a-priori how many times a given challenge can be accessed in the future before erasure. Notice that the erasibility property alone just allows an erasure function, but not a guarantee under which it has to be applied in the future. Note that, due to their programmability, PlayPUFs can, for example, be used to realize count-limited certificates [SvDO+06]. In addition, the performance evaluation and security analysis of PLayPUF are provided in the paper as well.

3. Utilizing the unique feature of memristors, we propose two physical erasable PUFs, which can erase one or a group of challenges without affecting the other challenges. Both proposed physical erasable PUFs, by introducing reconfigurability to individual underlying physical structure, overcome the one-time erasure limitation of the existing erasable PUFs. The two proposed physical erasable PUFs have their own advantages: mrSHIC-PUF allows one to erase CRPs one by one, but the total number of CRPs is limited, and thus we have to limit its readout speed. Erasable mrSPUF is a strong PUF, which provides larger CRP space, so its readout speed/throughput can be selected as needed. However, its erasure operation has to erase a group of CRPs (all the CRPs created by one column of memristor cells), so when it is used in a protocol, one needs to carefully select the CRPs used in the protocol.

1.2 Organization of This Paper

Section 2 presents a formal definitional framework of PUFs and its variants. After that, programmable logically erasable PUFs and memristor based physical erasable PUFs are introduced in Section 3 and 4, respectively. This paper concludes in Section 5.

2 A Formal Definitional Framework of PUFs

Intuitively, a silicon Physical Unclonable Function (PUF) is a fingerprint of a chip, that

Manufacturing Resistance: leverages process manufacturing variation to generate a unique function taking "challenges" as input and generating "responses" as output, which

HW Unclonability: 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

SW Unclonability: 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).

2.1 Stateless PUFs

A formal (ideal) definition of a PUF 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, it can output $\perp$. We model P_λ by an associated function $g[P_\lambda] : C_\lambda \to R_\lambda$ 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 $> 2^{-\lambda}$: For all λ, $c \in C_\lambda$, and ppt algorithms $\mathcal{A}$,

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

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. □

We argue that this definition captures the essence of PUFs for our purpose:

Measurement Noise: Definition 1 does not model measurement noise in the PUF itself. In practice one may need to correct noise by for example using a fuzzy extractor [DRS04, FMR13, HRvD+17, JHR+17]. We assume that interface circuitry for noise correction is included in the physical objects, and therefore we model combined PUFs + interfaces (which Definition 1 calls PUFs) as ideal functionalities without noise. We also assume that interface circuitry is included which expands inputs to the physical object to challenges to the PUF which are separated by some minimum distance so that produced responses become uncorrelated (notice that some PUF designs have correlated CRPs, e.g. arbiter PUFs [GCvDD02b], so a pre-hash in the PUF interface can seperate CRPs as desired [GCvDD02a]).

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 [GDC+08] which has a fused secret key K and implements $g(c) = Enc_K(c)$ for some semantically secure encryption scheme. The unclonability defined in Definition 1 does not formalize HW unclonability (in our applications we do not assume physical attacks which can break HW unclonability) but formalizes SW unclonability by only considering an adversary with black-box access to the physical object, i.e., the adversarial algorithm $\mathcal{A}$ has adaptive access to an oracle $G_{g,c}$ which represents the physical object as a black box. Notice that manufacturing resistance is covered by giving $\mathcal{A}$ access to an oracle M_λ which represents the manufacturing process. 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 [RvD13].

Software Unclonability: Definition 1 explicitly states that the object can 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 bits to be SW cloned as long as at least λ bits entropy are guaranteed).

Strong PUFs vs Weak PUFs: By convention, if the challenge space is too large to be exhaustively enumerated by an adversary, we call it a strong PUF. Otherwise, it is called a weak PUF. Notice that, our Definition 1 captures the properties of both strong and weak PUFs, because adversaries are required to predict an *unseen* CRP with an advantage of $poly(\lambda)$ adaptively chosen CRPs for learning. In order for Definition 1 to work, we need the CRPs of a PUF to be uncorrelated, see **Measurement Noise** for a detailed discussion.

Trusted Computing Base: Outside the TCB of a PUF 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. In the remainder of this proposal we implicitly assume such an extension to Definition 1.

2.2 Stateful PUFs

Due to the existence of strong adversaries who can look into the internal digital state of a circuit, we only assume tamper-resistant non-volatile memory in PUFs with state. 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 $poly(\lambda)$ 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 $\mathcal{A}$,

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

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.

Notice that the special case where the tamper-resistant state is always $s_{current} = s_{new} = \epsilon$ in Definition 2 is equivalent to Definition 1. For this reason next definitions will be based on families of PUFs with tamper-resistant state of the more general Definition 2.

2.3 Erasable PUFs

To fix key exchange in the PUF re-use model, as depicted in Figure 1, Rührmair and van Dijk pointed out that a PUF should be strengthened with other complementary features, such as making the CRPs "erasable" [vDR14]. To make this possible a PUF must have a form of non-volatile state to enable this erasure operation. In [RJA11], 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 3. [Erasable PUFs] A family of 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 3), 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.

Physical Erasable PUFs: In the above definition algorithm g's behavior can be the result of pure physics and engineering properties of the PUF in which case there is no digital non-volatile state $s_{current}$; instead of a digital state the physics properties of the engineered PUF itself somehow remembers what has been erased (set $\mathcal{E}$) and what has not been erased. When talking about physical erasable PUFs we mean exactly this kind. We introduce the our physical erasable PUF design (in terms of Definition 3) in Section 4.

Programmable Logical Erasable PUFs If we do rely on non-volatile tamper-resistant (but not tamper-evident) state $s_{current}$, then we do not necessarily rely on the PUF design itself; we create erasability of challenges by exploiting $s_{current}$. A logic circuitry which interfaces with the PUF-core interprets $s_{current}$ and in essence implements an access control policy. Not surprisingly, it turns out that $s_{current}$ can therefore be used to implement a more generalized form of erasable PUFs, coined programmable logical erasable PUFs introduced in Section 3.

2.4 (Partially) Reconfigurable PUFs

Unlike the research on erasable PUFs, another related PUF variant, called reconfigurable PUFs, attracted attention in the community. A variety of reconfigrable PUF designs have been proposed. They introduced reconfigurability either logically by adding a digital interface [KKvDL+11, LP11, EKvdL11], or physically by exploiting some physical features of special materials [KSS+09, ZKC+14, GRAS+15a, Che15, SRK+18]. However, they all reconfigure and, as a result, erase the entire CRP space at once. So, they all fall into the category of fully reconfigurable PUFs.

On the spectrum between erasable PUFs and fully reconfigurable PUFs, we notice that some PUFs can be partially reconfigurable, meaning they allow users to erase a subset of the entire CRP space without affecting the other CRPs, and after erasing a set of CRPs, reconfigure the responses corresponding to the erased challenges to new random responses. As explained in the introduction, for our physical erasable PUF designs to be practical (i.e. not deplete CRPs space too fast), we need reconfigurability property in addition to erasability. A definition of *partially reconfigurable/erasable PUFs* is similar to Definition 3, but for a specific list of subsets of CRPs. In particular, this partially reconfigurable/erasable PUF is of interest to us, because it allows us to build a strong physical erasable PUF, which has a much larger CRP space. This concept is demonstrated in Section. 4.4 by the design of erasable mrSPUFs.

3 Programmable Logical Erasable PUF

We extend Definition 3 towards programmable logical erasable PUFs: We propose to implement an erasability functionality 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 allows an application to program how many times a response for a given challenge can be extracted.

Definition 4. [Programmable Logical Erasable PUFs or PLayPUFs] A family of 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:

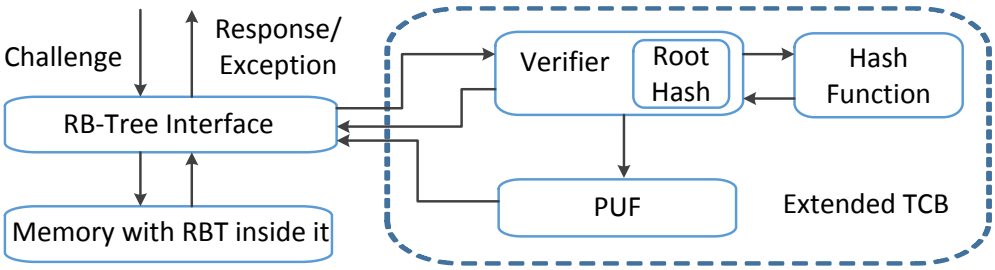


Figure 2: The entire system of Programmable Logically Erasable PUF

- 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 $(\perp, s_{new})$. □

Notice that $E(c) = 0$ indicates that the response corresponding to c should be erased. Algorithm $\bar{g}$ has oracle access to $E(.)$ and outputs the empty response if the corresponding challenge should have 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 PUFs with tamper-resistant state implies that if a response to a challenge has been erased, 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.

For completeness we notice that the logically reconfigurable PUF in [KKvDL+11] was the first to use tamper-resistant state for the purpose of reconfiguring *all* the CPRs together.

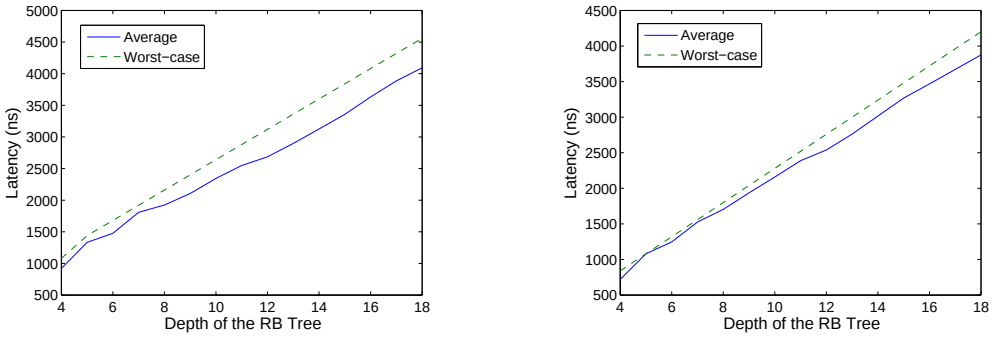
3.1 Implementation

We propose to merge a Red-Black Tree [CLR+01, Bay72] and an Authenticated Search Tree [BLL00] 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 the hashes of each of the 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 [CLR+01] 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 $Hash(c)$ and its most recent counter value $E(c)$, see Definition 4. The TCB will check whether $E(c) \neq 0$ in which case it allows a response for c to be computed.

Figure 2 shows the proposed design. It consists of a public software interface with public memory/state and a hardware TCB which contains the PUF functionality based on



(a) Initialization (b) Extraction

Figure 3: The average and worst-case latency for serving one challenge request with respect to the depth of the RB Tree.

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 hash value of a used challenge c, 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 $Hash(c)$, counter and the hash values stored in its children. The hashes are used to prove non-existence or integrity and freshness.

Intuitively, the complete working is 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. More details of the interaction between the untrusted part and the TCB is provided in Appendix A, together with the pseudocodes in Appendix B.

3.2 Evaluation

The proposed architecture has been implemented assuming an "ideal" noiseless strong PUF (we only assume adversaries with black box access and do not assume adversaries with side channel information, hence, the attacks of [TDF+14] does not apply). 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 [MvOV96, AES01]. We measured the performance of the proof of concept for challenge initialization (each challenge adds a new node in the tree) and response extraction (all challenges are existing in the tree) separately. Figure 3 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 $O(lg(n))$, where n is the number of nodes in the tree.

4 Physical Erasable PUFs based on Nanotechnology and Memristors

In this section, we investigate two physical erasable PUF realizations by exploiting unique properties realized in nano devices. We describe two erasable PUF constructions where a physical erase operation is irreversible and cannot be controlled by any party. We also allow free access to arbitrarily characterize CRPs of these two erasable PUF constructions while retaining their security properties.

First, we briefly introduce the memristor—the fundamental nano device chosen for realizing physically erasable PUF constructions—and the nano crossbar; then we presents two memristor and nano crossbar enabled physical erasable PUFs.

4.1 Memristors and Crossbars

A memristor is a two terminal non-volatile nano memory element. The memristor switches between high resistance state (HRS) and low resistance state LRS by applying a relatively large negative/positive potential difference ($V_{\text{SET}}/V_{\text{RESET}}$ as shown in Fig. 4 (c)) between the bottom electrode and top electrode, marked as '+' in Fig. 4 (b). The growth and disruption of filamentary conductive paths inside of insulating dielectrics are responsible for this switching behavior.

The HRS and LRS are typically treated as two logic states for storing digital information [YSS13, WS15]. To read out the logic states, a small voltage as illustrated in Fig. 4 (c) is applied and the current is sensed and compared with a reference current, if the readout current is higher, a LRS is determined, otherwise, a HRS is determined. Notably, for electric-field-induced bipolar switching, a small electric-field corresponding to a small potential difference across the memristor is inadequate to move filaments to change its resistance [YSS13]. Therefore, a small readout potential difference applied across the top and bottom electrode will not disturb its resistance. The non-volatility of a memristor relies on the fact that the resistance of the memristor remains unchanged when power is turned off. The memristor is a promising NVM candidate due to its smaller footprint, faster switching speed, higher endurance, lower power consumption and longer retention time.

Memristors are increasingly considered to be integrated with a simple crossbar architecture for memory applications. The two-terminal memristive device based crossbar array offers opportunities such as 3D integration, Field Effect Transistor (FET) fabrication compatibility, low power operation, and memory with a resistive nature rather than the traditional capacitance-based storage. A crossbar array—see Fig. 5 (a)—comprises of two layers of parallel electrodes that are crossed perpendicularly, they act as word-lines and bit-lines respectively. A memristor at each crosspoint acts as a switch, which can be programmed to the low resistance state (LRS) or high resistance state (HRS) representing either a logic "1" or "0".

Although compact and 3D stackable crossbar array memory structure enables storing super high information content (SHIC) in a small area, integrating a switching memristor into the crossbar array faces a challenge posed by sneak path (leakage) current that prevents the correct state (LRS and HRS) readout of individual memristors in large-size crossbars. The source of sneak-path currents is shown in Fig. 5 (a). We can see that, besides the desired read current, there are many current paths—one such current path is shown—flowing into the selected bit-line blurring the desired read current. Notably, the sneak path currents need to pass through at least three cells in the array. In extreme case shown in Fig. 5 (a), all memristors are in LRS except the selected one, we can see that sneak path currents are dominant in this context, because the HRS/LRS ratio are always high, eg., 100 [JKN+14].

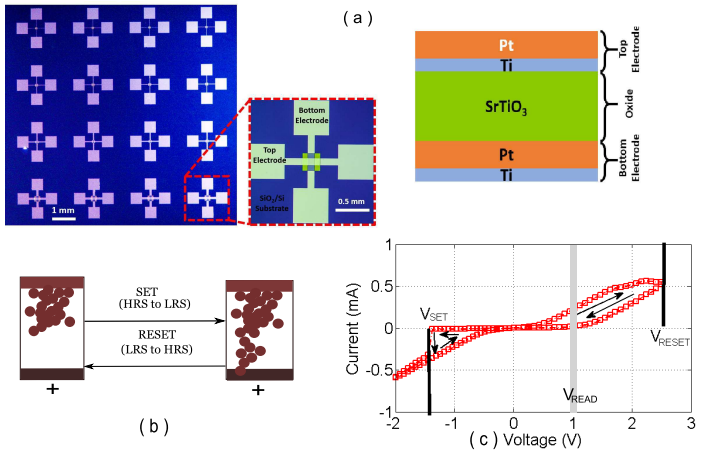


Figure 4: (a) Photomicrograph and cross-sectional view of fabricated memristors. (b) RESET and SET illustration of the memristor. RESET switches the memristor from LRS to HRS and is performed by applying a positive voltage V_{RESET}. SET switches the memristor from HRS to LRS and is performed by applying a negative voltage V_{SET}. (c) The current-voltage characteristic of a memristor we fabricated where $V_{\text{SET}} = -1.5$ V and $V_{\text{RESET}} = 2.5$ V.

To enable large-size and purely passive crossbar arrays to create high density information storage capability, there are a number of solutions that can be adopted [GKAS+16]. For example, a memristor with self-rectifying behavior significantly suppresses the sneak path currents. For memristors, such as that in Fig. 4, cannot suppress sneak path currents by themselves, a one-selector-one-memristor structure (1S1M) [JKN+14] can be adopted to greatly suppress sneak path currents while keeping the high density property of a crossbar array. The selector can be simply treated as a special diode that exhibits two directional volatile switching with large resistance ratio, high turn on current and steep turn on slope as depicted in Fig. 5 (c) [JKN+14].

4.2 Random Variations

A memristor stores information as different resistance states, the resistance is readout by a small applied voltage without disturbing its resistance. However, the readout margin—the capability of distinguishing two logical states, '0'/'1'—among different states is influenced by resistance variations. As for memristors, the resistance variation is not only from variations in geometry—eg. thickness, doping—determined by uncontrollable fabrication process variations but also from cycle-to-cycle (C2C) variation due to the random locations of filaments in the memristor—these metal filaments are formed and disrupted during HRS/LRS reprogramming [YSS13, WS15, GRAS+16]. In other words, the HRS/LRS resistance of a memristor varies once it is reprogrammed. As a consequence, these undesirable variations decrease the readout margin, and hence result in performance degradation when the memristor is used as a memory element. Therefore, device engineers always try to mitigate such variations.

Conversely, security applications embrace truly random variations. The scaling down to nano region provides more randomness to build physical unclonable functions. The high information density yields more response bits from a limited area. Moreover, the C2C variation is a unique property of memristors compared with CMOS devices. Notably, the conductivity in HRS is dominated by the tunneling across nano-gaps, therefore, slight location variations of nano-gaps converts to significant resistance variations in HRS. To

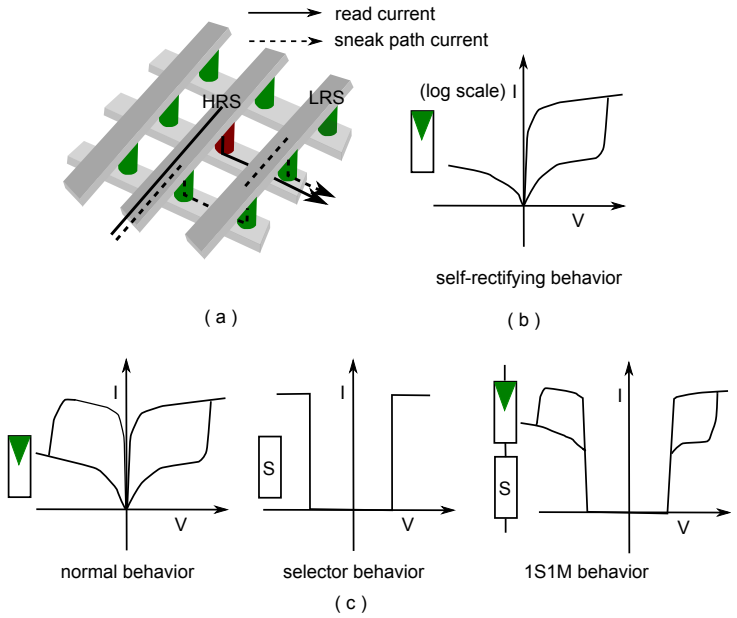


Figure 5: (a) Crossbar architecture. (b) The IV-characteristic of a memristor with self-rectifying behavior to suppress sneak path currents. (c) The IV-characteristics of a normal memristor—as in Fig. 4 (c), a two terminal selector, and the 1S1M structure, respectively. The 1S1M significantly suppresses the sneak path currents while still allowing the serially connected memristor being easily programmed, hence retains the high density of the crossbar without sacrificing the performance of the memristor.

validate the obvious C2C variation in HRS, we fabricated a number of memristors and tested them. A 50 nm thin film of $SrTiO_3$ is deposited on a $Pt/Ti/SiO_2$ (50:10:300 nm) pre-patterned Si substrate using RF magnetron sputtering at room temperature, from a stoichiometric ceramic target. Top Pt/Ti (50:10 nm) electrodes are fabricated by three-step photolithography/lift-off processes and deposited by using electron beam evaporation at room temperature. A detailed description of fabrication is in [NWK+15]. The photomicrograph of fabricated memristors are shown in Fig. 4 (a). The characterization of devices was performed by pulse transient measurements using a sourcemeter (Agilent 2912A). The current-voltage characteristic of a memristor from measurements is shown in Fig. 4 (c). From our measurements, we can easily observe C2C variations in Fig. 6, where the memristor exhibits different resistance values after each programming cycle.

Foregoing the high information density offered by crossbar arrays, the variations introduced from process variations and C2C variations enable the *physical erasable PUF* designs. In the following, we present two different erasable PUF realizations that meet the properties of physical erasable PUFs defined in Section 2.3.

4.3 Erasable memristor SHIC-PUF (mrSHIC-PUF)

The SHIC-PUF [RJB+11] is a weak PUF in terms of the number of CRPs, but it is resilient to modeling attacks because: i) its response bits are generated from independent entropy sources; ii) its higher information density offered by the use of a nano crossbar array; iii) the full characterization of all CRPs of a SHIC-PUF is prevented by intentionally slowing down readout speed. In SHIC-PUF, a diode is employed at the crosspoint of a crossbar and therefore cannot be reprogrammed. To enable CRPs to be erasable, a breakdown operation is introduced to the diode to change its IV (current-voltage) characteristics [RJA11].

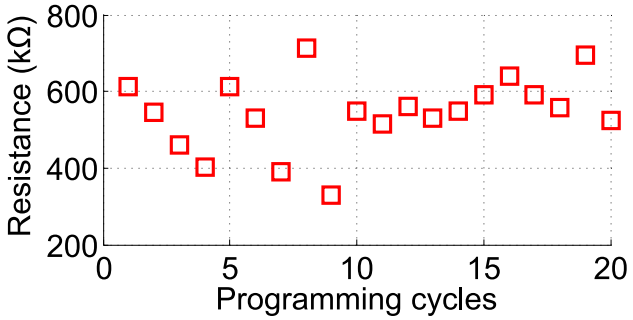


Figure 6: Cycle-to-cycle (C2C) variation. The R_{OFF}/HRS variation of an individual memristor for 20 cycles. Data is experimentally obtained from our fabricated memristor in Fig. 4. A factor of two in C2C variation can be observed. Note that even larger C2C variations have been reported (Fig. 8 in [CCZ+15]).

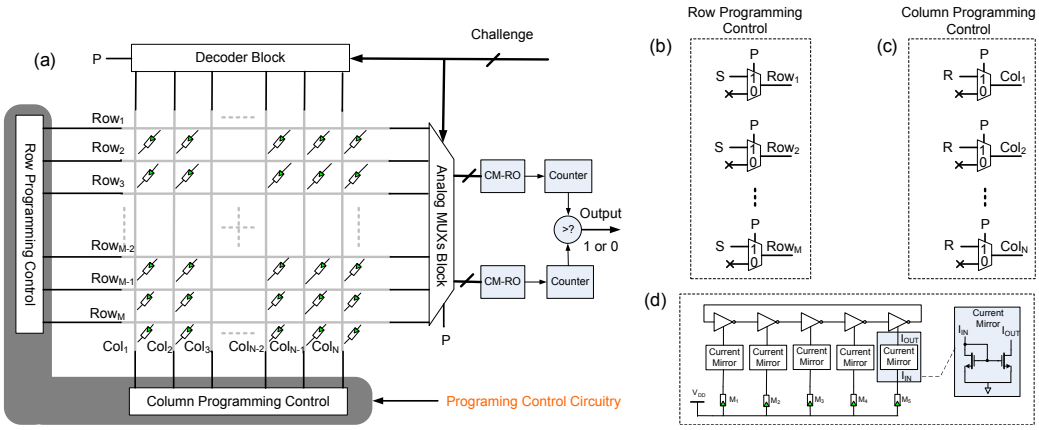


Figure 7: mrSPUF architecture. (a) Simplified mrSPUF architecture. All the memristors are in the LRS/HRS state. The shaded programming control circuit comprises of the row programming control circuitry in (b) and column programming control circuitry in (c), which is employed to program memristors in the nanocrossbar array before it acts as a PUF and facilitates reconfigurability of mrSPUF by subsequent reprogramming to refresh CRPs of mrSPUF to transform it into a new PUF instance. In contrast to the programming control circuit, the top decoder block and left analog multiplexers block, CM-ROs and counters enables the stimulation by a challenge and the extraction of a corresponding response. A challenge encoded as a vector of binary values (bits) is used to provide the address bits for both the analog multiplexers block and the decoder block. (d) CM-RO. Each current mirror starves only an inverter in the RO structure, where the bias memristor for each current mirror, M_i, is selected from the nanocrossbar array.

However, this erasure is only possible one-time for a given diode. In addition, the sneak path currents are significantly increased as the rectification behavior of the diode reduces after breakdown operation; consequently, preventing the readout of information to enable further response evaluations.

Using one-selector-one-memristor (1S1M) structure to replace the diode in the crossbar of the SHIC-PUF [RJB+11] realizes a memristor enabled physical erasable SHIC-PUF (mrSHIC-PUF) by exploiting C2C variations. This PUF architecture inherits the strong PUF property of the SHIC-PUF and further enables physically erasable individual CRPs. The mrSHIC-PUF avoids deterioration in readability of individual memristors due to sneak path currents after performing the erasure and further overcomes the one-time erasure limitation in [RJA11].

To erase a single CRP, one can simply reprogram the memristor in the 1S1M structure. For example, if the HRS resistance is treated as the entropy source to generate a response bit, then to erase such a response, one can SET the memristor from HRS to LRS first, then RESET it from LRS to HRS as illustrated in Fig. 4 (b). Due to the C2C variations, the current resistance in HRS is different from the previous resistance in HRS as shown by our measurements in Fig. 6. Consequently, a subsequent response from the same challenge—address of the nano crossbar—becomes unpredictable even if the previous response is known. Most notably, the erase operation is achieved by altering a physical property of a memristor, and the erasure is irreversible.

The mrSHIC-PUF also benefits from the high density of the nano crossbar and the intentionally slow readout speed inheriting from SHIC-PUFs. Considering a readout speed of 100 bits/second as in [RJB+11], while allowing uninterrupted readout for an adversary, 2.3×10^{10} bits of information is needed to be stored in a passive nano crossbar to ensure a period of more than ten years to readout all CRPs. This volume of information is indeed practical by using a high performance selector with experimentally reported selectivity of 10^{10} in the 1S1M structure [JKN+14]. In [JKN+14], 4Mb 1S1M nano crossbar has been experimentally demonstrated. Under the 100 bits/second intentionally slow readout speed, it will already take more than 11 hours to acquire all CRPs by an adversary.

The next realization of a physically erasable PUF mitigates the requirement of an intentionally slow readout speed of a mrSHIC-PUF, which requires careful constructions of the nano crossbar to ensure adequately slow readout speed.

4.4 Erasable mrSPUF

4.4.1 mrSPUF Architecture

The mrSPUF shown in Fig. 7 was proposed by Gao *et al.* [GRAS+15a]. The mrSPUF architecture combines a nanocrossbar and current mirror controlled ring oscillators (CM-ROs) to realize not only a strong PUF but also a reconfigurable PUF, where the variations exploited is sourced not only from fabrication but also from programming operations induced C2C variations. Initially, all memristors are programmed to HRS. In the mrSPUF, $2 \times i$ memristors in the same column are selected. Each selected memristor is then used to control the current in a single current mirror and consequently to starve the current in each inverter in the ring oscillator, to achieve a *current starved ring oscillator* structure called a current mirror controlled ring oscillator (CM-RO) as illustrated in Fig. 7 (a) and (d). Therefore, the delay time of an inverter in a CM-RO is a direct function of the current through the memristor selected to starve it, in turn the delay time is related to the resistance of the memristor. Consequently, the oscillation frequency of a CM-RO is a function of i selected memristors and measured using a counter. Subsequently, a response bit is generated by comparing the outputs from the two counters.

In general mrSPUF has two CM-ROs, each RO has i inverters, and $2 \times i$ memristors are randomly selected to configure each CM-RO. So the total number of CRPs ($\mathrm{NT}_{\mathrm{CRP}}$)

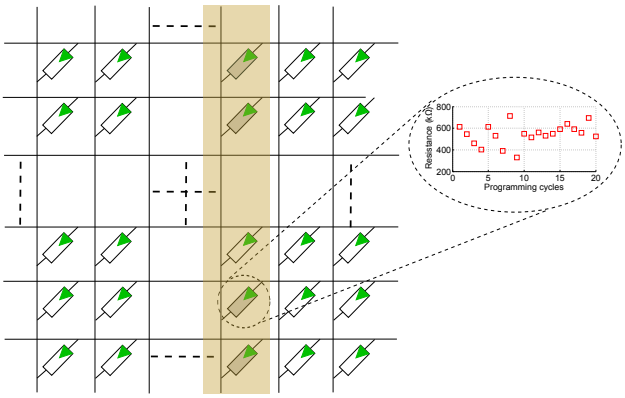


Figure 8: Each memristor can be reprogrammed and exhibits the inherent C2C variations, which makes the mrSPUF reconfigurable. All the randomly selected memristors (marked as gray) to configure the CM-RO in mrSPUF are from the same column (shadowed column). This feature enables a mrSPUF to act as an erasable PUF. Individual memristors selected by a challenge corresponding to a specific response can be re-programmed. Given the C2C variations of the memristors, reprogramming will ensure that the new resistance in HRS/LRS is, once again, non-deterministic. Hence, the response bits determined by comparisons of memristors' resistance are unpredictable after reprogramming.

in this configuration is estimated as:

$$\mathrm{NT}_{\mathrm{CRP}} = \frac{N \times \binom{M}{i} \times \binom{M-i}{i}}{2} = O(N \cdot M^i \cdot (M-i)^i) \tag{2}$$

where N is the number of columns, M is the number of rows. Therefore, the total number of CRP grows exponentially in i making mrSPUF a strong PUF. According to equation 2, one can also calculate the number of inverters in CM-RO or the nanocrossbar array size needed based on the desired number of CRPs. For example, using a nanocrossbar array with equal number of rows and columns ($N = M = 100$) and a 5-stage ($i = 5$) CM-RO, we can acquire 2.1811×10^{17} CRPs. Then it is estimated that it will take over 690 years for an attacker to fully readout all possible CRPs [GRAS+15a].

The mrSPUF is initially demonstrated to be a reconfigurable PUF [GRAS+15a]. To enable the mrSPUF to behave as a reconfigurable PUF, additional circuitry is not necessary. A mrSPUF can be reconfigured by reprogramming all the memristors in the nanocrossbar.

4.4.2 Partial Reconfigurability

Switching the memristors selected by a challenge from the HRS state to the LRS, and then switching it back from the LRS state to the HRS state again, using the programming control circuitry shown in Fig. 7, will endow physical erasability to a mrSPUF. However, the memristors selected by each challenge are not as independent as in the mrSHIC-PUF. However, note that each challenge applied to mrSPUF only selects $2 \times i$ memristors in the same column as visualized in Fig. 8. This feature enables the mrSPUF to act as a partially erasable/reconfigurable PUF even if the CRPs are not totally independent of each other because the reconfiguration of response bits extracted from one column does not influence the response bits generated from other columns.

4.4.3 Security

We notice that mrSPUF produces responses in the same way as a ring oscillator PUF, and a ring oscillator PUF can be perfectly modeled by sorting algorithms which sort the

frequencies produced by all possible ring oscillator pairs [RSS+10]. As noted in the security analysis in the original mrSPUF paper [GRAS+15b], an adversary with knowledge of mrSPUF structure and full control of the PUF still have two main difficulties in modeling a *single* column mrSPUF (note that each column produces independent CRPs, so this analysis can be easily extended to a multi-column mrSPUF): firstly, an adversary is not able to directly exploit the responses to compare the frequencies of two CM-ROs which share any common memristor cells, because two non-overlapping groups of memristor cells in the same column have to be selected for producing a response; secondly, even if the adversary can somehow figure out the sorted order of the delay values corresponding to each individual memristor cells, it still cannot always successfully predict the responses, because knowing the order of individual delay values does not directly teach how to compare two sums of delay values from two non-overlapping groups. A more detailed security analysis can be found in [GRAS+15b]. The best known attack to perfectly model a *single* column mrSPUF is to sort all frequencies produced by $\binom{M}{i}$ possible ring oscillators, which yields a worst-case CRP complexity of $O(M^i \cdot i \cdot logM)$ given full control of the PUF [CLR+01, RSS+10].

In addition, even if an adversary is able to use an unknown attack to model mrSPUF, we can still limit the modeling capacity of the adversary by reconfiguring CRPs *before* each protocol invocation and *after* each protocol ending. In this way, what needs to be (or has been) modeled will have been erased completely.

5 Conclusion

Motivated by the "PUF re-use model", we investigated two ways of implementing erasable PUFs, namely programmable logically erasable PUFs and physical erasable PUFs. The programmable logically erasable PUF (PLayPUF) acts as an interface to deny the access to erased CRPs. As an additional property, we gain programmability for free, which means that users can precisely define a-priori how many times a given challenge can be accessed in the future before erasure. The proposed physical erasable PUFs utilize the cycle-to-cycle variation of memristors to alter one or one group of CRPs physically. Both of them advance the current state-of-the-art in erasable PUFs research, which is crucial in realizing secure PUF-based cryptographic protocols.

Moreover, to formalize our study and locate erasable PUFs together with other PUF variants in the overall PUF landscape, we introduce a formal definitional framework of PUFs, which captures the properties of strong/weak PUFs with/without state and with/without erasability.

Acknowledgements

This research was supported in part by the Australian Research Council Discovery Program (DP140103448), AFOSR MURI under award number FA9550-14-1-0351, and an NSF grant CNS-1617774 "Self-Recovering Certificate Authorities using Backward and Forward Secure Key Management". Ulrich Rührmair gratefully acknowledges funding by the PICOLA project of the German Bundesministerium für Bildung und Forschung (BMBF).

References

[AES01] AES, NIST. Advanced encryption standard. *Federal Information Processing Standard, FIPS-197*, 12, 2001.

[Bay72] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta informatica*, 1(4):290–306, 1972.

[BFSK11] Christina Brzuska, Marc Fischlin, Heike Schröder, and Stefan Katzenbeisser. Physically uncloneable functions in the universal composition framework. In *Advances in Cryptology CRYPTO 2011*, pages 51–70. Springer, 2011.

[BLL00] Ahto Buldas, Peeter Laud, and Helger Lipmaa. Accountable certificate management using undeniable attestations. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 9–17. ACM, 2000.

[CCZ+15] Bing Chen, Fuxi Cai, Jiantao Zhou, Wen Ma, Patrick Sheridan, and Wei D Lu. Efficient in-memory computing architecture based on crossbar arrays. In *IEEE International Electron Devices Meeting*, pages 17–5. IEEE, 2015.

[Che15] An Chen. Utilizing the variability of resistive random access memory to implement reconfigurable physical unclonable functions. *IEEE Electron Device Letters*, 36(2):138–140, 2015.

[CLR+01] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[DRS04] Yevgeniy Dodis, Leonid Reyzin, and Adam Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In *Advances in cryptology Eurocrypt 2004*, pages 523–540. Springer, 2004.

[EKvdL11] Ilze Eichhorn, Patrick Koeberl, and Vincent van der Leest. Logically reconfigurable pufs: Memory-based secure key storage. In *Proceedings of the sixth ACM workshop on Scalable trusted computing*, pages 59–64. ACM, 2011.

[FMR13] Benjamin Fuller, Xianrui Meng, and Leonid Reyzin. Computational fuzzy extractors. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 174–193. Springer, 2013.

[GCvDD02a] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Controlled physical random functions. In *Computer Security Applications Conference, 2002. Proceedings. 18th Annual*, pages 149–160. IEEE, 2002.

[GCvDD02b] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Silicon physical random functions. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 148–160. ACM, 2002.

[GDC+08] Blaise Gassend, Marten van Dijk, Dwaine Clarke, Emina Torlak, Srinivas Devadas, and Pim Tuyls. Controlled physical random functions and applications. *ACM Transactions on Information and System Security (TISSEC)*, 10(4):3, 2008.

[GKAS+16] Yansong Gao, Omid Kavehei, Said F Al-Sarawi, Damith C Ranasinghe, and Derek Abbott. Read operation performance of large selectorless cross-point array with self-rectifying memristive device. *INTEGRATION, the VLSI journal*, 54:56–64, 2016.

[GRAS+15a] Yansong Gao, Damith C Ranasinghe, Said F Al-Sarawi, Omid Kavehei, and Derek Abbott. Memristive crypto primitive for building highly secure physical unclonable functions. *Scientific Reports*, 5, art. no. 12785, 2015.

[GRAS+15b] Yansong Gao, Damith C Ranasinghe, Said F Al-Sarawi, Omid Kavehei, and Derek Abbott. Memristive crypto primitive for building highly secure physical unclonable functions supplementary information s1. 2015.

[GRAS+16] Yansong Gao, Damith C Ranasinghe, Said F Al-Sarawi, Omid Kavehei, and Derek Abbott. Emerging physical unclonable functions with nanotechnology. IEEE Access, 4:61–80, 2016.

[HRvD+17] Charles Herder, Ling Ren, Marten van Dijk, Meng-Day Yu, and Srinivas Devadas. Trapdoor computational fuzzy extractors and stateless cryptographically-secure physical unclonable functions. *IEEE Transactions on Dependable and Secure Computing*, 14(1):65–82, 2017.

[JHR+17] Chenglu Jin, Charles Herder, Ling Ren, Phuong Ha Nguyen, Benjamin Fuller, Srinivas Devadas, and Marten van Dijk. Fpga implementation of a cryptographically-secure puf based on learning parity with noise. *Cryptography*, 1(3):23, 2017.

[JKN+14] Sung Hyun Jo, Tanmay Kumar, Sundar Narayanan, Wei D Lu, and Hagop Nazarian. 3D-stackable crossbar resistive memory based on field assisted superlinear threshold (FAST) selector. In *IEEE International Electron Devices Meeting*, pages 6–7, 2014.

[KKvDL+11] Stefan Katzenbeisser, Ünal Kocabaş, Vincent van Der Leest, Ahmad-Reza Sadeghi, Geert-Jan Schrijen, and Christian Wachsmann. Recyclable pufs: Logically reconfigurable pufs. *Journal of Cryptographic Engineering*, 1(3):177–186, 2011.

[KSS+09] Klaus Kursawe, Ahmad-Reza Sadeghi, Dries Schellekens, Boris Skoric, and Pim Tuyls. Reconfigurable physical unclonable functions-enabling technology for tamper-resistant storage. In *Hardware-Oriented Security and Trust, 2009. HOST'09. IEEE International Workshop on*, pages 22–29. IEEE, 2009.

[LLG+04] Jae W Lee, Daihyun Lim, Blaise Gassend, G Edward Suh, Marten van Dijk, and Srinivas 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*, pages 176–179. IEEE, 2004.

[LLG+05] Daihyun Lim, Jae W Lee, Blaise Gassend, G Edward Suh, Marten van Dijk, and Srinivas Devadas. Extracting secret keys from integrated circuits. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 13(10):1200–1205, 2005.

[LP11] Yingjie Lao and Keshab K Parhi. Reconfigurable architectures for silicon physical unclonable functions. In *Electro/Information Technology (EIT), 2011 IEEE International Conference on*, pages 1–7. IEEE, 2011.

[MvOV96] Alfred J Menezes, Paul C van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[NWK+15] Hussein Nili, Sumeet Walia, Ahmad Esmaielzadeh Kandjani, Rajesh Ramanathan, Philipp Gutruf, Taimur Ahmed, Sivacarendran Balendhran, Vipul Bansal, Dmitri B Strukov, Omid Kavehei, et al. Donor-induced performance tuning of amorphous $SrTiO_3$ memristive nanodevices: Multistate resistive switching and mechanical tunability. *Advanced Functional Materials*, 25:3172–3182, 2015.

[OSVW13] Rafail Ostrovsky, Alessandra Scafuro, Ivan Visconti, and Akshay Wadia. Universally composable secure computation with (malicious) physically uncloneable functions. In *Advances in Cryptology–EUROCRYPT 2013*, pages 702–718. Springer, 2013.

[PRTG02] Ravikanth Pappu, Ben Recht, Jason Taylor, and Neil Gershenfeld. Physical one-way functions. *Science*, 297(5589):2026–2030, 2002.

[RJA11] Ulrich Rührmair, Christian Jaeger, and Michael Algasinger. An attack on PUF-based session key exchange and a hardware-based countermeasure: Erasable PUFs. In *Financial Cryptography and Data Security*, pages 190–204. Springer, 2011.

[RJB+11] Ulrich Rührmair, Christian Jaeger, Matthias Bator, Martin Stutzmann, Paolo Lugli, and György Csaba. Applications of high-capacity crossbar memories in cryptography. *Nanotechnology, IEEE Transactions on*, 10(3):489–498, 2011.

[RSS+10] Ulrich Rührmair, Frank Sehnke, Jan Sölter, Gideon Dror, Srinivas Devadas, and Jürgen Schmidhuber. Modeling attacks on physical unclonable functions. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 237–249. ACM, 2010.

[Rüh10] Ulrich Rührmair. Oblivious transfer based on physical unclonable functions. In *Trust and Trustworthy Computing*, pages 430–440. Springer, 2010.

[RvD13] Ulrich Rührmair and Marius van Dijk. Pufs in security protocols: Attack models and security evaluations. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 286–300. IEEE, 2013.

[SD07] G Edward Suh and Srinivas Devadas. Physical unclonable functions for device authentication and secret key generation. In *Proceedings of the 44th annual Design Automation Conference*, pages 9–14. ACM, 2007.

[SRK+18] Soubhagya Sutar, Arnab Raha, Devadatta Kulkarni, Rajeev Shorey, Jeffrey Tew, and Vijay Raghunathan. D-puf: An intrinsically reconfigurable dram puf for device authentication and random number generation. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(1):17, 2018.

[SvDO+06] Luis FG Sarmenta, Marten van Dijk, Charles W O'Donnell, Jonathan Rhodes, and Srinivas Devadas. Virtual monotonic counters and count-limited objects using a tpm without a trusted os. In *Proceedings of the first ACM workshop on Scalable trusted computing*, pages 27–42. ACM, 2006.

[TDF+14] Shahin Tajik, Enrico Dietz, Sven Frohmann, Jean-Pierre Seifert, Dmitry Nedospasov, Clemens Helfmeier, Christian Boit, and Helmar Dittrich. Physical characterization of arbiter pufs. In *Cryptographic Hardware and Embedded Systems–CHES 2014*, pages 493–509. Springer, 2014.

[vDR14] Marten van Dijk and Ulrich Rührmair. Protocol attacks on advanced puf protocols and countermeasures. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 351. European Design and Automation Association, 2014.

[WS15] H-S Philip Wong and Sayeef Salahuddin. Memory leads the way to better computing. *Nature nanotechnology*, 10(3):191–194, 2015.

[YSS13] J Joshua Yang, Dmitri B Strukov, and Duncan R Stewart. Memristive devices for computing. *Nature nanotechnology*, 8(1):13–24, 2013.

[ZKC+14] Le Zhang, Zhi Hui Kong, Chip-Hong Chang, Alessandro Cabrini, and Guido Torelli. Exploiting process variations and programming sensitivity of phase change memory for reconfigurable physical unclonable functions. *IEEE Transactions on Information Forensics and Security*, 9(6):921–932, 2014.

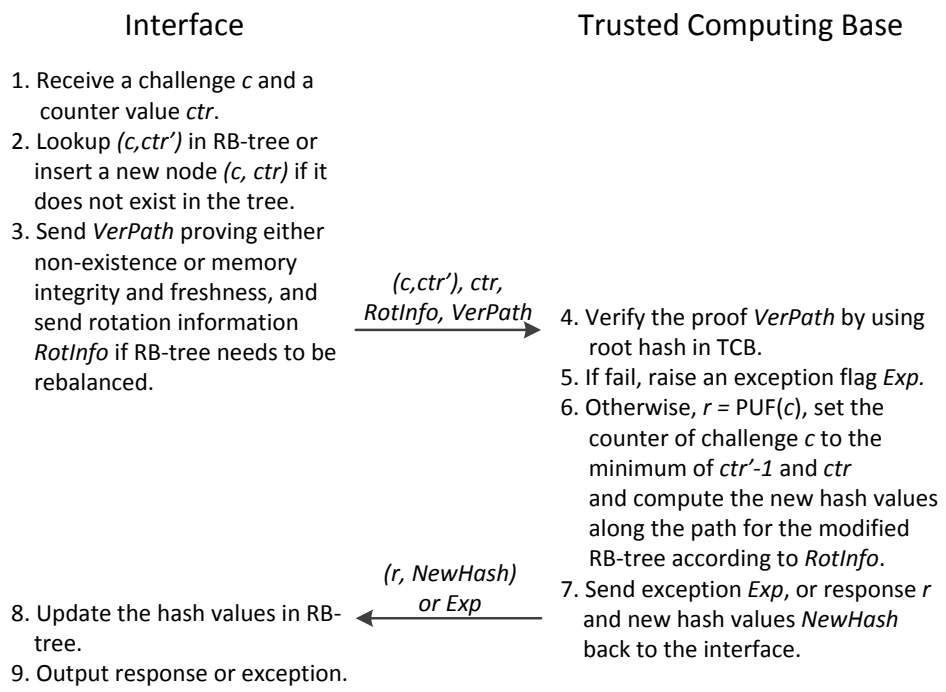


Figure 9: The protocol between software information and hardware TCB of a Programmable Logically Erasable PUF.

A 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. 9. We remind the reader that the TCB part consist of a tamper-resistant and private PUF together with tamper-resistant additional circuitry which includes non-volatile state, see Fig. 2.

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 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

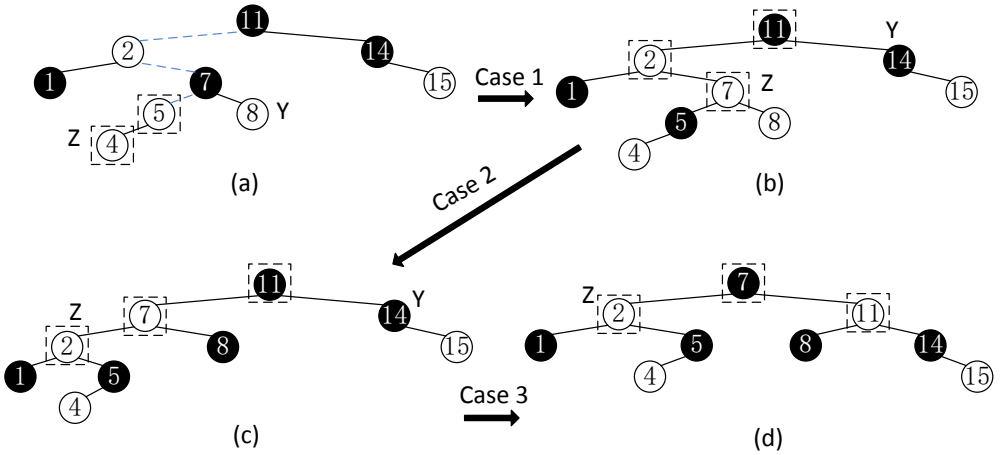


Figure 10: Insertion of a new node 4.

explain a detailed example in Fig. 10. 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. 10 depicts an example of consecutive operations in Red-Black Tree Insert-Fixup, see [CLR+01]. (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 [CLR+01] 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 [CLR+01] 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 $VerPath$.

B Pseudocodes of PLayPUF Implementation

Algorithm 1 RB-Tree-Interface. (Note that this code is modified based on **RB-Insert** in [CLR+01]. 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 [CLR+01]. The lines added by us are indicated by *. Also, the pseudocodes of **Left-Rotate** and **Right-Rotate** can be found in [CLR+01]).

```
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

procedure UPDATE-HASH(Length of Proof N, Proof *VerPath*, Query *Query*, Counter Value *ctr*, Rotation-Information *RotInfo*)
  if $Query.ctr = \infty$ **then** ▷ Newly added node
    $j = N - 1$
  else ▷ Existing node
    $j = N - 2$
  end if
  $Query.ctr = min(Query.ctr - 1, ctr)$
  $i = 0$
  $NewHash[i++] = Hash(Query.ch \,||\, Query.ctr \,||\, Query.lhash \,||\, Query.rhash)$
  $ch = Query.ch$
  if $RotInfo \neq NULL$ **then**
    while $i < (RotInfo.case1 + RotInfo.case4)$ **do** ▷ Case 1 and 4
      $(ch, NewHash, i, j) \leftarrow$ HASH-NO-ROTATION$(ch, VerPath, NewHash, i, j)$
    end while
    if $RotInfo.case2 == 1$ **then** ▷ Case 2
      $NewHash[i] = Hash(VerPath[j-1].ch \,||\, VerPath[j-1].ctr \,||\, VerPath[j-1].shash \,||\, NewHash[i-1])$
      $NewHash[i+1] = Hash(VerPath[j-2].ch \,||\, VerPath[j-2].ctr \,||\, VerPath[j].shash \,||\, VerPath[j-2].shash)$
      $ch, NewHash[i+2] = VerPath[j].ch, Hash(VerPath[j].ch \,||\, VerPath[j].ctr \,||\, NewHash[i] \,||\, NewHash[i+1])$
    else if $RotInfo.case3 == 1$ **then** ▷ Case 3
      $NewHash[i] = Hash(VerPath[j].ch \,||\, VerPath[j].ctr \,||\, VerPath[j].shash \,||\, NewHash[i-1]$
      $NewHash[i+1] = Hash(VerPath[j-2].ch \,||\, VerPath[j-2].ctr \,||\, VerPath[j-1].shash \,||\, VerPath[j-2].shash)$
      $ch, NewHash[i+2] = VerPath[j-1].ch, Hash(VerPath[j-1].ch \,||\, VerPath[j-1].ctr \,||\, NewHash[i-2] \,||\, NewHash[i-1])$
    else
      (same as **then** clauses for case 2 and 3 with the order of two children's hash values exchanged)
      $i = i + 3$
      $j = j - 3$
    end if
  end if
  while $j \geq 0$ **do** ▷ No Fixup
    $(ch, NewHash, i, j) \leftarrow$ HASH-NO-ROTATION$(ch, VerPath, NewHash, i, j)$
  end while
  $root = NewHash[i-1]$
  Return *NewHash, root*
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