Software Techniques to Combat Drift in PUF-based Authentication Systems

Michael S. Kirkpatrick and Elisa Bertino
Department of Computer Science
Purdue University
West Lafayette, IN 47907, USA
Email: {mkirkpat,bertino}@cs.purdue.edu

Abstract

Physically unclonable functions (PUFs) provide a mechanism for uniquely identifying a hardware device based on the intrinsic variations of physical components. Common applications for PUFs include generating or binding cryptographic keys in a secure manner. In recent work, though, we have examined the possibility of incorporating PUFs directly into certain cryptographic protocols as part of an access control mechanism. These protocols rely on the ability of the PUF to produce an identical binary value at every attempt.

One challenge with using PUFs in these authentication protocols is that device behavior can change as the device ages. This change is called drift*. Although short-term drift can be handled by applying error-correcting codes, long-term drift requires a more robust solution. In this work, we propose two software-based schemes for addressing drift. Our approaches, using the Feige-Fiat-Shamir identification scheme and Merkle hash trees, work either by responding to detected drift or by attempting to prevent drift from affecting the PUF mechanism. We also examine the security guarantees and trade-offs involved in adopting either approach.*

I. Introduction

When multiple computing devices are manufactured, the physical nature of the world introduces slight variations that are beyond the control of the designer. For instance, even if two semiconductors are manufactured from the same silicon wafer, wires designed to be the same will probably differ in width by a few nanometers; microscopic differences in the surface of the silicon will induce almost trivial variations in the curviture of lines.

While most work in computer engineering focuses on how to produce identical behavior despite these variations, recent work on physically unclonable functions (PUFs) proposes the opposite approach. That is, as these unique characteristics are uncontrollable and inherent to the physical device, quantifying them can produce an unforgeable identifier.

From a high-level perspective, a PUF can be defined as a mapping of challenges and responses. If an input challenge C_i is presented to a PUF on a particular device, the response will always be R_i. Presenting the same C_i to the PUF on a different device will produce $R_i' \neq R_i$. Furthermore, if two PUFs exist on the same device, the output responses will be different, just as if the PUFs were on different devices. Additionally, in an ideal PUF, the mapping between C_i and R_i is unpredictable and random. For instance, if C_i and C_j differ in only a single bit, knowledge of R_i does not reveal usable information to predict R_j.

A common application of PUFs is to implement secure storage of cryptographic keys. If a key K is created externally to the device but needs to be stored locally, the device can store $X = K \oplus R_i$ for some PUF response. As the bit string that makes up R_i is random (and works essentially a one-time pad), X can be stored anywhere without endangering the secrecy of K. When the key is needed at run-time, the key is reconstructed as $K = X \oplus R_i$. In other settings, the PUF can also be used to generate keys by mapping R_i directly to some key K. For instance, computing the SHA-256 hash of R_i can produce a key for use in AES encryption.

As an alternative to using PUFs to store cryptographic keys, we have recently proposed a mechanism for incorporating PUFs directly into cryptographic protocols [1], [2]. The goal of these protocols is to give the device a mechanism to prove its identity (by demonstrating knowledge of the PUF behavior) as part of an access

request, while avoiding the performance and organizational overhead of public key cryptography. Thus, the objective of our research is to produce secure protocols designed to exploit the physical properties of PUFs, especially for use in custom system-on-chip or embedded devices.

While the ideal of a PUF is very attractive, there exists a critical problem with their long-term use: As a device ages, its physical characteristics change. This change is often called *drift*. These changes are evident even in the short-term, as PUF responses are based on noisy data; the use of error-correcting codes is a common approach to producing the necessary consistency in the short-term.

However, as the device is continuously used and the PUF is repeatedly executed, the drift may become so great that the error-correcting codes are no longer effective. Specifically, error-correcting codes have an inherent threshold regarding how many bits they can correct; once the drift causes the raw PUF response to exceed this allowable difference, the official PUF response R_i is unretrievable. In the case of our protocols, the result is that it would then become impossible for the device to identify itself.

Thus, the problem we are trying to address in this paper is how to combat the effects of drift in an authentication mechanism. By authentication, we are specifically referring to the ability of the device to prove its identity based on the PUF it possesses.

Our solution is to propose two software-based approaches. The first is based on a detection & response mechanism. The idea is that the device detects that drift is starting to push the limits of what the error-correcting codes can fix, but the device is still able to produce the correct result. As a result of the detected drift, the device triggers a recommitment protocol with the server to update information about its PUF.

The second approach is to prevent drift from affecting the authentication mechanism by continuously updating the PUF commitment. That is, the lifetime of each challenge-response pair is so short that the device simply cannot age sufficiently to affect the PUF response.

The rest of this paper is formatted as follows. Sections II and III provide background material on related work and PUFs. Sections IV provides an overview of our approaches, which we detail in Sections V, VI, and VII. We summarize the trade-offs of each approach in Section VIII and conclude in Section IX.

II. Related Work

A number of papers have proposed the design and implementation of PUFs [3], [4], [5], [6], [7], [8]. The applications for PUFs proposed in these works, though, have focused on storage or creation of traditional keys. Given the advantages of PUFs for these applications, [9] and [10] explored the possibility of designing a secure processor with an integrated PUF. The advantage of such a scheme is that keys and other sensitive data are never transmitted across an unsecured bus. Additionally, researchers have also begun exploring techniques for uniquely identifying other types of devices beyond just circuits; for example, [11], [12], and [13] propose techniques for identifying RFID instances, while [14] identifies a fingerprinting technique for CDs.

Besides our previous work [1], [2], [15] and [16] are perhaps the most similar to our objectives, as these schemes examine the use of PUFs for purposes other than binding cryptographic keys. However, the former focuses on binding software in a virtual machine environment, whereas the latter focuses on authenticating banking transactions. Our principle application scenario is to use device identification as part of an access request in low-power embedded systems. Furthermore, our current work focuses exclusively on how to handle the problem of drift in these scenarios.

As drift has long been understood to exist, recent research in the area has focused on how to model the aging of components and circuits [17], [18], [19], [20]. For example, these approaches involve the use of accelerated aging techniques in an attempt to extract parameters that model the changes that occur. These techniques offer the promise of estimating the effect of aging on PUF responses, but cannot provide definite predictions. Although we can speculate that coupling modeling with error-correcting codes may offer a technique to counter drift, it is not clear if this approach will guarantee accuracy.

Our software techniques are built on three existing techniques. First, we rely heavily on the use of Reed-Solomon (RS) codes [21] to provide a consistent response from the raw PUF output; given an input of k data symbols, RS(n,k) can correct up to $(n-k)/2$ errors.

Next, our recommitment protocol involves a variant of the Feige-Fiat-Shamir [22] identification scheme. This scheme is a zero-knowledge protocol, meaning that observing the transcript of an authentication session reveals no useful information about a secret. The advantage of applying a zero-knowledge proof to a PUF-based authentication system should be evident; given that a PUF is inherently bound to the device, it cannot be revoked when compromised, so it must be closely guarded.

It is important to contrast our work with similar work that uses biometrics to generate cryptographic keys [23]. In that work, the authors apply an XOR of bits to the raw biometric data. In the context of biometrics, this extra step is necessary to allow for revokability. Furthermore, biometrics are actually public, as fingerprints, etc., are left on every device we touch. Thus, the XOR step binds

the biometric to the token. In contrast, we argue that this step is unnecessary for PUF-based authentication. First, to revoke a PUF, the server can simply stop using the relevant challenge. Next, the PUF behavior is not public, and only exists at run-time. Hence, this XOR does not add to the security of the system.

The third technique we employ is that of the Merkle hash tree. A Merkle tree is a binary tree such that each node stores the hash of its children. Assuming a strong cryptographic hash is used, the root of the tree can be made public without endangering the secrecy of the leaf nodes, which hold the actual secret values. Someone with knowledge of a leaf and certain internal nodes can prove it by revealing the values, which can be used to reconstruct the public root value.

III. Physically Unclonable Functions

The fundamental idea of PUFs is to create a random pairing between a challenge input C and a response R. The random behavior is based on the premise that no two instances of a hardware design can be identical. That is, one can create a PUF by designing a piece of hardware such that the design is intentionally non-deterministic. The physical properties of the actual hardware instance resolve the non-determinism when it is manufactured. For example, the length of a wire in one device may be a couple of nanometers longer than the corresponding wire in another device; such differences are too small to be controlled and arise as natural by-products of the physical world.

Consider the circuit design in Figure 1, which describes how to generate a 1-bit PUF from ring oscillators (ROs). A RO consists of a circular circuit containing an odd number of not-gates; this produces a circuit that oscillates between producing a 1 and 0 as output. To produce a 1-bit PUF, the output of two ROs pass through a pair of multiplexors (MUX) into a pair of counters storing the number of fluctuations between the 0 and 1 output.

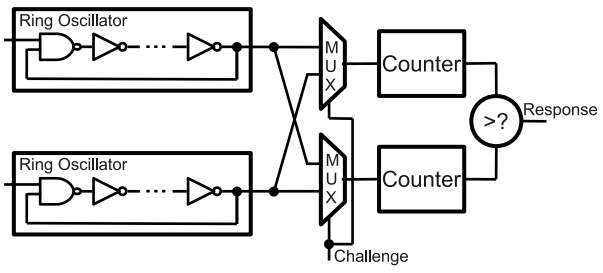

Fig. 1. A sample 1-bit PUF based on ring oscillators

As it is nearly impossible for the ROs to operate at the same frequency, with almost certain probability, the counters will hold different values after the same amount of time. The PUF result is 1 if the counter on top holds a greater value, 0 otherwise. The role of the challenge in a 1-bit RO PUF is to flip the MUX. That is, flipping the challenge bit will flip the PUF result.

Larger PUFs can be created by using a pool of ROs, and the challenge is used to select random pairings for comparison. One issue that designers of PUFs need to consider is the degree of linkage between response. For instance, flipping only a single bit in the challenge input may select nearly identical pairs of ROs to be selected; the result would be similar responses. To prevent these linkages, the PUF result is often run through a cryptographic hash function; the responses then differ greatly.

One problem with PUFs is that the non-deterministic nature often leads to inconsistent behavior. Specifically, the response usually involves noisy data; applying the same challenge to the same PUF may produce a response in which a small number of bits are sometimes 1 and sometimes 0. The use of error-correcting codes, such as Reed-Solomon, can solve this inconsistency in the short term. However, as the device ages and its physical properties change, the number of bit errors can increase as a result of the drift. Addressing this problem is the aim of this paper.

IV. Approaches to Combating Drift

We identify three primary mechanisms for addressing PUF drift in authentication systems:

- **Detection & Recommitment.** Our first approach is to use a threshold scheme to detect drift as it begins to occur. Once the drift has been observed, a secure protocol is triggered to update the PUF behavioral commitment automatically and remotely.
- **Prevention.** Instead of waiting for drift to occur, another approach is to prevent drift from becoming a factor by reducing the lifetime of any particular challenge-response pair.

Our detection mechanism consists of a multi-tiered approach to error-correcting codes that are designed to produce identical results. As drift occurs, the results of applying these codes would differ and make the presence of drift known. For our recommitment protocol, we start with an authentication protocol based on that of Feige, Fiat, and Shamir [22]. We then propose a variation for securely updating the secret values used. To illustrate our prevention approach, we propose a scheme that applies Merkle hash trees to secure the PUF commitment.

V. Drift Detection

The physical nature of PUFs introduces noise to the observed behavior, and deployment of PUFs for authentication purposes must account for this variation. For example, short-term fluctuations in the execution environment may result in a small number of bits to differ from the expected binary value. In the case of RO-based PUFs, a pair of oscillators may operate at nearly identical speeds; the "faster" oscillator for a single PUF execution may alternate as a result. However, digital authentication systems rely on the ability to reproduce an exact value with no bit errors.

The use of error-correcting codes is a well-established technique for producing a consistent result from noisy data. Reed-Solomon (RS) codes [21] are an example of such a scheme that is frequently used. In RS codes, a block of input symbols are appended with a number of symbols called a syndrome. Based on the mathematical properties of the codes, the syndrome allows for the correction of a number of errors in the input data.

RS codes depend on two parameters, n and k. The first parameter, n, is defined to be $n = 2^m - 1$, where m is the size of each symbol and n is the total number of symbols in the input data an the syndrome. As bytes are prevalent in digital systems, $m = 8$ is a commonly used value, so $n = 255$. The other parameter, $k < n$, is the number of input data symbols. Once the syndrome has been constructed, the mathematical properties of RS codes allow for the correction of up to $(n-k)/2$ corrupted symbols in the input data. Hence, RS(255,223) denotes a code where $n = 255$ and $k = 223$; this code takes 223 input symbols, produces a syndrome of 32 symbols, and can correct up to 16 corrupted symbols.

In a previous work [2], we built a PUF-based authentication system that produced 64 bits of output from a simplistic design of 128 pairs of ring oscillators. Although our design was simplistic and produced a small output, a trivial approach[1] to producing a larger value would be to chain the results of successive PUF executions. For example, one could build a 1024-bit key by polling the PUF 16 times. In our experiments, we observed an average of 0.2 bits of errors per execution. Over 16 executions, we would accumulate, on average, 3.2 incorrect bits per 1024-bit key. By interpreting the 1024-bit key as a sequence of 128 bytes, one could pad the PUF result with 95 bytes of 0 and apply a RS(255,223) to produce a consistent result.

[1]Obviously, there are disadvantages to this approach over a more sophisticated design, but this basic design works for illustrative purposes.

A. Multi-tiered Error-Correction for Drift Detection

Given the resilience of RS codes for correcting variations in the PUF data, this approach has been proposed quite often in works on PUFs. It seems intuitive, then, to adapt this technique to serve as a drift detection mechanism. That is, instead of storing a single syndrome for the PUF response, the device would store two, where the k values differ. For instance, in a setting where RS(255,223) successfully captures the short-term variations in the PUF behavior, the device could also store a RS(255,191) code. Clearly, both codes will work for a 1024-bit PUF result.

As we are assuming that RS(255,223) is sufficient, decoding the output of a single PUF execution with both codes *should* produce identical results. This fact should be obvious, as RS(255,191) can correct 32 corrupted bytes, whereas RS(255,223) can only correct 16. Hence, if the results ever differ, there is a high likelihood that the disparity indicates the presence of drift. That is, the behavior of the device has changed sufficiently that there are more than 16 bytes that differ from the expected result.

While it may be possible that applying the RS(255,191) code does not produce the correct result, we are assuming this is not the case. That is, drift is a slow process, implying that the number of errors in the raw PUF response increase gradually. Thus, if drift is detected, the RS(255,191) code can still produce the correct result for use in a recommitment protocol as described in the next section. In an extreme setting requiring high security, if the RS(255,191) code cannot recover the correct PUF response, the drift could trigger a self-destruct mechanism. However, we will assume that the RS(255,191) code is successful.

VI. Recommitment Protocols

Consider the Feige-Fiat-Shamir protocol illustrated in Figure 2. In this cryptographic protocol, P indicates the PUF-enabled client and S denotes the server. When the device is initially registered, the PUF is executed using a set of challenges $C_1, C_2, \ldots, C_m$. The device then computes the modular squares $R_1^2, R_2^2, \ldots, R_m^2$ (all mod N)[2], and these values are stored on the server. (To simplify the discussion, all operations are performed (mod N), and we will simply omit this piece of notation henceforth.) The security of this protocol relies on the intractability of efficiently computing modular square roots. Hence, an

[2]An important caveat is that it should hold that the greatest common divisor $GCD(R_i, N) = 1$ for all $1 \leq i \leq m$. If this is not the case, either that challenge could be discarded, or one could divide R_i by $GCD(R_i, N)$. The extended Euclidean algorithm can compute this value efficiently. As this problem can be easily addressed, we will simply assume that $GCD(R_i, N) = 1$ for all $1 \leq i \leq m$.

attacker cannot easily retrieve R_i from R_i^2 for any value of i, let alone all of them. Thus, an attacker with access to the server cannot build a model of the PUF.

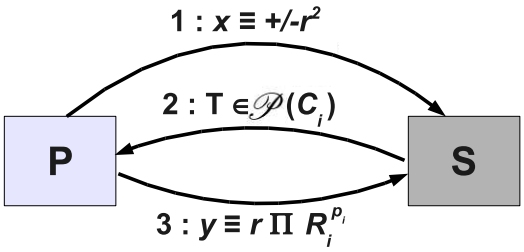

Fig. 2. Using PUFs in the Feige-Fiat-Shamir protocol

In the protocol, P selects a random value r and computes its modular square. To ensure the zero-knowledge properties of the protocol, P sends either r^2 or $-r^2$. S selects a random set T of the challenges $C_1, C_2, \ldots, C_m$. That is, $\mathscr{P}(C_i)$ denotes the power set of the challenges. To prove knowledge of the secret values $R_1, R_2, \ldots, R_m$, P computes the product of the responses corresponding to the challenges in T. Formally, $\forall C_i \in T$, $p_i = 1$, and $\forall C_i \notin T$, $p_i = 0$. This product is multiplied by the random r to produce the value y, which is sent to S. S accepts the proof if and only if $y^2 = x$ or $y^2 = -x$.

This protocol offers a number of advantages for use with PUF-enabled devices. First, the intractability of modular square roots prevents an attacker with access to the server from learning the values of $R_1, R_2, \ldots, R_m$, preventing the attacker from modeling the PUF. Second, the probability of guessing the correct response is negligible, providing a high level of assurance that P's identity is correct. Third, the zero-knowledge properties of the protocol ensure that an attacker who can observe the transcript of an authentication session learns nothing of value. That is, the protocol does not leak any information regarding the PUF response. Finally, the protocol does not use any modular exponentiation. The client only needs to perform multiplication, which makes this approach nice for low-power embedded devices.

A. Updating the Challenge-Response Pairs

Given this usage of Feige-Fiat-Shamir, the difficulty becomes how to provide a secure mechanism to update the commitment to the challenge-response pairings. Our aim is that the mechanism should provide the following guarantees:

- **Authenticity.** Only P can update its pairings. That is, an attacker should not be able to forge an unauthorized update for P.
- **Integrity.** The new pairs (C_i, R_i) should be protected from tampering by a man-in-the-middle attack.
- **Secrecy.** An eavesdropper should not learn information regarding the new commitment pairings.

To achieve these security goals, we propose the protocol shown in Figure 3. Unlike the traditional Feige-Fiat-Shamir protocol, this new protocol requires sending two sets of challenges T and T'. The first set is used as before. Hence, P will compute y just as in Figure 2. However, instead of sending y to S, P uses y^2 as a seed to create a one-time-use symmetric encryption key. We use $E_{otk}(m)$ to indicate message m encrypted under this one-time key. In this case, the message is the new pairings $(C_i' : R_i'^2)$ $\forall C_i \in T'$. Note that, as P sent either x or $-x$, S may need to attempt the decryption twice, switching the sign accordingly.

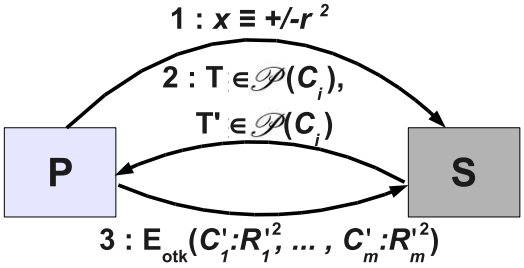

Fig. 3. Updating the challenge-response pairs with Feige-Fiat-Shamir

B. Security Analysis

In analyzing this protocol, we assume a probabilistic polynomial-time adversary $\mathcal{A}$. As such, assume that such an attacker cannot produce y or break the symmetric key encryption scheme with greater than negligible probability. We also assume that the current set of challenge-response pairings is unknown to the attacker.

Under these assumptions, authenticity holds, because the ability to produce the correct one-time key relies on knowledge of y. That is, a correctly formatted message can only have been produced by the legitimate P. Similarly, integrity and secrecy both hold as a result of the strength of the symmetric key cryptography algorithm.

Note that we are not considering the case of denial-of-service as part of integrity. That is, an attacker could jam the communication signals between S and P, preventing the messages from transmission. Furthermore, an attacker could launch a simple permanent denial-of-service simply by destroying P. Clearly, addressing such an attack requires measures far beyond a protocol, and we do not consider them further.

VII. Drift Prevention

A Merkle hash tree [24] is a binary tree data structure, where each node stores the cryptographic hash of its children. To adapt this data structure for PUFs, consider the design in Figure 4, where adjacent leaves store the challenge-response pairs.

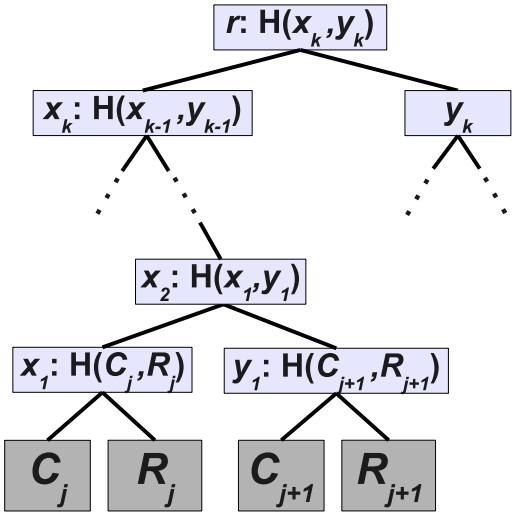

Fig. 4. A Merkle hash tree of PUF challenge-response pairs

Starting at the leaf nodes for C_j and R_j, we denote the parent nodes as x_i and the auxiliary node required for the hash as y_i. For example, the parent of C_j and R_j would be $x_1 = H(C_j, R_j)$. The parent of x_1 would store $x_2 = H(x_1, y_1)$, where y_1 stores the hash of another challenge-response pair. After constructing this hash tree, the root of the tree, denoted $r = H(x_k, y_k)$, is made public as a commitment to the values stored in the tree.

There are two challenges to using this data structure for PUFs. First, the PUF must have a way to prove knowledge of the structure of the tree underlying the public root value. A simple approach would be to pick a random challenge-response pair and reveal the values of x_i and y_i necessary to traverse the tree from the leaves to node. However, doing so leads to the second challenge, which is to minimize the amount of information leakage in the proof.

Clearly, the selected challenge-response pair would need to be discarded. As a result, the x_i values would need to be updated. Similarly, the public root value would need to change. From the perspective of designing a robust update protocol, the goals are similar to those in Section VI, namely authenticity, integrity, and secrecy. To accomplish these goals, consider the protocol in Figure 5.

Before delving into the details of the protocol, it is necessary to make explicit what values are known prior

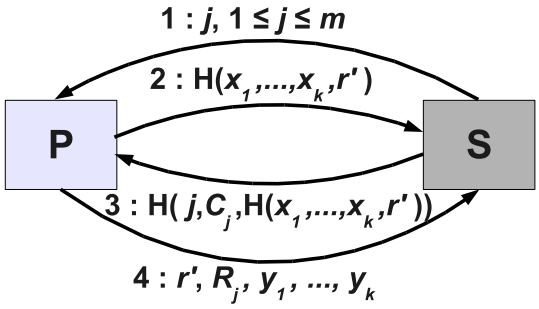

Fig. 5. A protocol for authenticating a PUF Merkle hash tree

to execution. S must maintain an array of the challenges $C_1, \ldots, C_m$, and these values must be private. As such, they are never transmitted except during initialization of the device. The root r is public and does not need to be protected. Our only stipulation is that, during initialization, a trusted channel exists so that the first value of r reported is correct for that device. Other than $C_1, \ldots, C_m$, and r, no secret value is stored at any time.

The protocol starts with S picking a random index j, $1 \leq j \leq m$. P then computes the relevant x_i and y_i values starting with the pair (C_j, R_j). The nature of the protocol requires this pair not be reused, so C_j is replaced with $H(C_j)$. For now, assume that this new challenge is used in the same location as the existing C_j. As a result of this update, all of the x_i values will need to be updated, and a new root r' will be produced. In step 2 of the protocol, P sends a commitment of $H(x_1, \ldots, x_k, r')$, where $x_1, \ldots, x_k$ represent the internal structure of the tree *prior to* authentication. This hash demonstrates knowledge of the existing tree structure (which authenticates P), but also binds the identification to the new root r' (preventing an attacker from inserting a new root value).

As we will detail in our security analysis, step 3 is critical. By creating a hash of j, C_j, and the hash from step 2, S is confirming that it has received the commitment from P. Furthermore, as only S and P know the challenge C_j, the hash must have been constructed by S. As such, step 3 provides assurance to P that S is the legitimate server. Once P receives this assurance, it reveals the new root value r', the PUF response R_j, and the intermediate values $y_1, \ldots, y_k$.

Using its private knowledge of C_j, S computes $x_1 = H(C_j, R_j), x_2 = H(x_1, y_1), \ldots, r = H(x_k, y_k)$. If this result matches the public root value, P's identity has been confirmed. Next, to ensure the integrity of the new root, S uses the reconstructed $x_1, \ldots, x_k$ values and the claimed r' to compute the hash reported in step 2. If all the hashes are correct, S accepts the new root r' and replaces

the challenge C_j with $H(C_j)$ for future authentication sessions.

A. Features and Variations

This scheme offers a number of interesting and attractive features. Using only cryptographic hash functions, it avoids any use of symmetric or public key encryption. Hence, it completely obviates the problem of key distribution. Furthermore, since the responses are only generated at run-time, *no secret value is ever stored on* P. In addition, assuming j is selected from an approximately uniform distribution, this approach inherently addresses the problem of drift, as the lifetime of any challenge-response pair is limited to m authentication sessions on average. Thus, assuming authentication occurs frequently enough that m sessions occur before aging effects take hold, the continuous updating performed in this protocol would prevent long-term drift from interfering with the authentication process.

Although these features are very attractive, we must highlight two minor shortcomings of the basic protocol and propose variations that improve these problems. First, the performance overhead for this scheme requires at least $O(m\ log\ m)$ hashes be computed each time. That is, the values $y_1, \ldots, y_k$ are determined by the subtrees rooted at each node. As such, if nothing is stored, the entire tree must be recomputed at run-time. The performance could be improved by storing a subset of the intermediate nodes; for instance, every left-child (except those whose children are challenge-response pairs) in the tree could be stored in a cache and retrieved as needed. Doing so would improve the performance to $O(log\ m)$ hashes, but requires storing a certain amount of somewhat secret data.

Next, if the same j is used every time, the $y_1, \ldots, y_k$ values will never change. One approach to prevent reuse of these values would be to shuffle the order of the leaf nodes and reconstruct the tree in its entirety each time; a simple shuffle would be to shift every challenge-response pair one position to the right. While these techniques will prevent reuse of the y_i values, we are not convinced that doing so is critical to the security of the system.

For an attacker to exploit the reuse of the y_i values, he must produce a PUF response R_j that will yield a sequence of $x_1, \ldots, x_k$ such that $H(x_k, y_k) = r$. Assuming a strong cryptographic hash is used used (*i.e.,* the hash is resilient against pre-image attacks), this attack is infeasible. Furthermore, with no knowledge of C_j, the attack becomes even more unlikely, as the attacker cannot know how to construct R_j such that $x_1 = H(C_j, R_j)$. Nevertheless, we discuss this issue for the sake of completeness.

B. Security Analysis

In analyzing this protocol, we assume a probabilistic polynomial-time adversary $\mathcal{A}$. Additionally, we assume H is a cryptographic hash function that is resilient against pre-image attacks. That is, an attacker is unable to produce x (or any part of it) upon observation of $H(x)$ with greater than negligible probability. Also, as in Section VI, we do not consider the problem of denial-of-service attacks. Our final assumption is that the challenges $C_1, \ldots, C_m$ are kept secret.

Given these assumptions, authenticity holds as a result of the security of the hash in step 2. That is, $H(x_1, \ldots, x_k, r')$ cannot be constructed without knowledge of $x_1, \ldots, x_k$, and these values depend directly on the PUF challenge-response mechanism. When S receives the data in step 4, it will use its stored challenge C_j, along with $R_j, y_1, \ldots, y_k$ to reconstruct the root value. If the result of hashing these values does not match the public value, the proof is not accepted. Similarly, S would use the data from step 4 to reconstruct the values $x_1, \ldots, x_k$, and validate the new root r' by computing the hash sent in step 2. Thus, a successful proof in which all values match could only originate from the PUF device.

Similarly, integrity holds because of the hash and the interleaving of dependencies between the steps. If j (step 1) is modified in transit, the hash in step 3 would be incorrect, as a computationally bound attacker could not forge such a hash. If the data in steps 2 or 3 are corrupted, P would detect the tampering after receiving step 3. Hence, any corruption in steps 1-3 would be detected and the protocol would be aborted before revealing any secret data.

Note that encryption of step 4 is unnecessary because of the integrity of the preceding steps. If any of $R_j, y_1, \ldots, y_k$ are corrupted, S would not be able to recompute the current public root value. If the r' is modified, S would not be able to recompute the hash sent in step 2. As steps 1-3 are known to be correct, S would detect any tampering in step 4 and request for P to resend the necessary data.

The secrecy guarantees of this protocol depend on the type of PUF being used. In our design, we are assuming an ideal PUF, in which revealing the pair (C_j, R_j) reveals no useful information regarding $(C_k, R_k)\ \forall\ j, k$. As an added measure, the challenge C_j is never actually revealed, so the attacker has no information on how to construct the response. Additionally, the strength of the cryptographic hash ensures that revealing the values $y_1, \ldots, y_k$ does not reveal information about any other pair (C_k, R_k).

However, if the PUF entails a linkage between responses, the protocol may reveal too much information about the PUF itself. This threat becomes even greater if the attacker has somehow managed to obtain the set of challenges. Thus, if the PUF deployed has such linkage

characteristics, this protocol should not be used.

VIII. Comparison of Approaches

The approaches proposed in the preceding sections offer a number of trade-offs. Table I summarizes the system requirements for comparison. On the client side, our detection & recommitment scheme requires additional storage for the extra RS(255,191) codes (one for each challenge-response pair), but offers lower computational overhead. There is no great disparity between the server-side performance or storage requirements. Hence, from a storage and computation perspective, the trade-offs favor the Feige-Fiat-Shamir detection & recommitment scheme.

Factor	Detection/Recommitment	Prevention
Client storage	RS(255,223) codes, RS(255,191) codes, $C_1, \ldots, C_m$	RS(255,223) codes, $C_1, \ldots, C_m$
Server storage	$(C_1, R_1^2), \ldots, (C_m, R_m^2)$	$C_1, \ldots, C_m, r$
Client computation	$O(m)$ modular multiplications, additional error-correction decoding, symmetric key encryption	$O(m\ log\ m)$ hashes (can be improved)
Server computation	$O(m)$ modular multiplications, symmetric key encryption	$O(log\ m)$ hashes
Encryption overhead	Symmetric key	none
Server knowledge of PUF response	$R_i^2 \pmod N\ \forall i$	none before authentication session
Lifetime of a PUF challenge-response pair	In use until drift occurs	m sessions on average

TABLE I. Summary of trade-offs between approaches

On the other hand, the Merkle-based prevention mechanism offers a number of advantages that are more difficult to quantify. First, the device does not require implementing symmetric key encryption, as it is never used. Next, the only *a priori* exposure of the PUF challenge-response pair is an aggregated hash of all such pairs, whereas the Feige-Fiat-Shamir scheme exposes the modular squares of every response. Hence, the Feige-Fiat-Shamir scheme offers more information for an attacker who attempts to model the PUF (although the attack is still infeasible under current cryptographic assumptions). Finally, the Merkle scheme greatly limits the lifetime of each challenge-response pair to a short time span. Besides preventing drift, this approach also reduces the attack surface; that is, the information learned by an eavesdropper will become obsolete very quickly. Therefore, these intangible benefits seem to favor the Merkle scheme. The choice of which scheme to deploy can only be made after considering the specifics of the application and the PUF used.

IX. Conclusions

The capability of deploying PUFs for long-term use in an authentication system relies on the ability of the device to produce the same result for an indeterminate amount of time. Although error-correcting codes can solve the problem of short-term variations in the PUF response, aging and drift can interfere with the device's ability to prove its identity.

We have identified two approaches to combating drift in PUF-based authentication systems. The first approach involves detecting the beginning of drift as it occurs and triggering a recommitment protocol in response. The second attempts to prevent drift from having an effect on the PUF behavior by restricting the duration of each challenge-response pair. Both mechanisms are software-based, and offer promise in extending the lifetime of the PUF-enabled device.

Deploying these approaches involves considering a number of trade-offs, which we have highlighted. The choice of the scheme to adopt requires careful consideration of the application scenario and the PUF used.

References

[1] M. Kirkpatrick and E. Bertino, "Physically restricted authentication with trusted hardware," in *The Fourth Annual Workshop on Scalable Trusted Computing (ACM STC '09)*, November 2009.

[2] M. S. Kirkpatrick, S. Kerr, and E. Bertino, "Enforcing physically restricted access control for remote files," *under submission*.

[3] K. Lofstrom, W. Daasch, and D. Taylor, "IC identification circuit using device mismatch," in *Solid-State Circuits Conference, 2000. Digest of Technical Papers. ISSCC. 2000 IEEE International*, 2000, pp. 372–373.

[4] G. E. Suh and S. Devadas, "Physcal unclonable functions for device authentication and secret key generation," in *Proceedings of the 44th IEEE Design Automation Conference (DAC)*. IEEE Press, 2007, pp. 9–14.

[5] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls, "Physical unclonable functions and public-key crypto for FPGA IP protection," in *International Conference on Field Programmable Logic and Applications*, 2007, pp. 189–195.

[6] ——, "FPGA intrinsic PUFs and their use for IP protection," in *Proceedings of the 9th Cryptographic Hardware and Embedded Systems Workshop (CHES)*, 2007, pp. 63–80.

[7] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas, "Silicon physical random functions," in *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS '02)*, 2002.

[8] ——, "Controlled physical random functions," in *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC)*, 2002.

[9] G. E. Suh, C. W. O'Donnell, and S. Devadas, "AEGIS: A single-chip secure processor," in *Elsevier Information Security Technical Report*, vol. 10, 2005, pp. 63–73.

[10] ——, "AEGIS: A single-chip secure processor," *IEEE Design and Test of Computers*, vol. 24, no. 6, pp. 570–580, 2007.

[11] S. Devadas, E. Suh, S. Paral, R. Sowell, T. Ziola, and V. Khandelwal, "Design and implementation of PUF-based "unclonable" RFID ICs for anti-counterfeiting and security applications," in *2008 IEEE International Conference on RFID*, 2008, pp. 58–64.

[12] B. Danev, T. S. Heydt-Benjamin, and S. Čapkun, "Physical-layer identification of RFID devices," in *Proceedings of the USENIX Security Symposium*, 2009.

[13] N. Saparkhojayev and D. R. Thompson, "Matching electronic fingerprints of RFID tags using the hotelling's algorithm," in *IEEE Sensors Applications Symposium (SAS)*, February 2009.

[14] G. Hammouri, A. Dana, and B. Sunar, "CDs have fingerprints too," in *Proceedings of the 11th Workshop on Cryptographic Hardware and Embedded Systems (CHES 2009)*, 2009, pp. 348–362.

[15] M. J. Atallah, E. D. Bryant, J. T. Korb, and J. R. Rice, "Binding software to specific native hardware in a VM environment: The PUF challenge and opportunity," in *VMSEC '08*. ACM, 2008.

[16] K. B. Frikken, M. Blanton, and M. J. Atallah, "Robust authentication using physically unclonable functions," in *Information Security Conference (ISC)*, September 2009.

[17] F. Marc, B. Mongellaz, C. Bestory, H. Levi, and Y. Danto, "Improvement of aging simulation of electronic circuits using behavioral modeling," *IEEE Transactions on Device and Materials Reliability*, vol. 6, pp. 228–234, June 2006.

[18] D. Lorenz, G. Georgakos, and U. Schlichtmann, "Aging analysis of circuit timing considering NBTI and HCI," in *15th IEEE International On-Line Testing Symposium (IOLTS)*, June 2009, pp. 3–8.

[19] S. V. Kumar and C. H. K. S. S. Sapatnekar, "Adaptive techniques for overcoming performance degradation due to aging in digital circuits," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, January 2009, pp. 284–289.

[20] W. Wang, V. Balakrishnan, B. Yang, and Y. Cao, "Statistical prediction of NBTI-induced circuit aging," in *9th International Conference on Solid-State and Integrated-Circuit Technology (ICSICT)*, October 2008, pp. 416–419.

[21] M. Riley and I. Richardson, "Reed-solomon codes," http://www.cs.cmu.edu/afs/cs.cmu.edu/project/pscico-guyb/realworld/www/reedsolomon/reed_solomon_codes.html, 1998.

[22] U. Feige, A. Fiat, and A. Shamir, "Zero knowledge proofs of identity," in *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, 1987, pp. 210–217.

[23] F. Hao, R. Anderson, and J. Daugman, "Combining crypto with biometrics effectively," vol. 55, no. 9, pp. 1081–1088, September 2006.

[24] R. C. Merkle, "A digital signature based on a conventional encryption function," in *CRYPTO '87: A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*. London, UK: Springer-Verlag, 1988, pp. 369–378.