

# Enforcing Physically Restricted Access Control for Remote Data

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

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

## ABSTRACT

In a distributed computing environment, remote devices must often be granted access to sensitive information. In such settings, it is desirable to restrict access only to known, trusted devices. While approaches based on public key infrastructure and trusted hardware can be used in many cases, there are settings for which these solutions are not practical. In this work, we define physically restricted access control to reflect the practice of binding access to devices based on their intrinsic properties. Our approach is based on the application of physically unclonable functions. We define and formally analyze protocols enforcing this policy, and present experimental results observed from developing a prototype implementation. Our results show that non-deterministic physical properties of devices can be used as a reliable authentication and access control factor.

## Categories and Subject Descriptors

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

## General Terms

Security

## Keywords

physically unclonable functions, applied cryptography, access control

## 1. INTRODUCTION

Controlled remote access to protected resources is a critical element in security for distributed computing systems. Often, some resources are considered more sensitive than others, and require greater levels of protection. Recent advances in access control [6, 1, 21] provide means to tighten the security controls by considering users' contextual factors. While these techniques offer more fine-grained control than traditional identity-based approaches, we desire an even stronger guarantee: Our goal is to provide a means by which access is granted only to known, trusted devices.

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

CODASPY'11, February 21–23, 2011, San Antonio, Texas, USA.  
Copyright 2011 ACM 978-1-4503-0465-8/11/02 ...\$10.00.

To achieve our aim, we had to address two separate issues. First, we required the ability to identify a device uniquely. That is, our scheme must be able to distinguish between two devices with software that is configured identically. Second, we had to establish a mechanism for encrypting the data for access by only the identified device.

A naïve approach to this problem would be to apply authentication mechanisms at the network and transport layers, for instance Challenge-Handshake Authentication Protocol (CHAP), Transport Layer Security (TLS), or Internet Protocol Security (IPsec). However, these solutions fail to provide our desired security guarantees in three ways. First, they differentiate based on stored data, *e.g.*, cryptographic keys. If this data is leaked, these solutions can be broken. Our approach, however, does not rely on the security of data stored on the client.

Second, these approaches are too coarse-grained, granting or denying access below the application layer. That is, our solution allows a server program to selectively grant access to subsets of data based on the unique hardware of the remote device. Existing approaches cannot provide this flexibility.

The third and final shortcoming of these basic approaches is that they can be completely bypassed by improper management and insider threats. In a recent report [35], the most common cause (48%) of data breaches was privilege misuse, which includes improper network configuration and malicious insider threats. In our approach, access control decisions are based on the physical properties of the remote devices themselves. While this does not completely eliminate insider threats, our solution does offer a higher level of defense against such insider threats.

Alternatively, one could rely on a public key infrastructure (PKI) using trusted platform modules (TPMs). While these approaches will work in traditional computing environments, our interests extend to environments for which TPMs are not available or PKI is considered to be too expensive. Specifically, we desire a solution that could also be deployed in low-power embedded systems. In these scenarios, the computing power required for modular exponentiation can quickly exhaust the device's resources. Our approach relies on a cryptographic scheme that offers similar guarantees as PKI, but with less computation required.

Our solution is based on the use of physically unclonable functions (PUFs) [14, 15]. PUFs rely on the fact that it is physically impossible to manufacture two identical devices. For example, two application-specific integrated circuits (ASICs) can be manufactured on the same silicon wafer, using the same design. However, a circuit in one ASIC may execute faster than the equivalent circuit in the other, because the wire length in the first is a nanometer shorter than the second. Such variations are too small to control and can only be observed during execution. PUFs quantify

these variations as challenge-response pairs, denoted  $(C, R)$ , that are unique to each particular hardware instance. A robust PUF is unpredictable, yet consistent for a single device. It is also unforgeable, as the physical variations that determine the PUF are too small to control.

Previous works on PUFs have focused on two areas. First, PUFs can be used to store cryptographic keys in a secure manner. Given a PUF pairing  $(C, R)$  and a key  $K$ , the device stores  $X = R \oplus K$ . In this case,  $R$  acts as a one-time pad, and  $X$  is a meaningless string of bits that can be stored in plaintext on a hard-drive. When the key is needed at a later time, the device again executes the PUF to get  $R$  and recovers the key as  $K = R \oplus X$ . The second use of PUFs is to generate cryptographic keys directly by mapping  $R$  to, for example, a point on an elliptic curve. In such a usage, the PUF does not have to store any data.

The advantages of employing PUFs for key generation and storage are subtle, and may be missed at first glance. First, note that no cryptographic keys are explicitly stored; the only data above that is ever stored is the value  $X$ , which is a random, meaningless bit string that reveals no information regarding the key  $K$ . A second advantage, which follows from the first, is that any key exists only at run-time. Furthermore, if the PUF is integrated into the processor itself, then the keys never even exist in main memory. Thus, PUFs offer very strong protections of cryptographic keys.

While these previous works assume a traditional cryptographic scheme is in place, we propose a new and unique direction for PUF research. That is, we propose incorporating the randomness of the PUF directly into an application-layer access request protocol. Our light-weight multifactor authentication mechanism, coupled with a dynamic key generation scheme, provides a novel technique for enforcing access control restrictions based on the device used.

The main contributions of our work can be summarized as follows.

- We propose the notion of physically restricted access control. That is, we propose integrating a device’s distinct characteristics directly into an access request.
- We define protocols for registering a device and making an access request, and present formal analyses of the security guarantees.
- We present a prototype implementation of our client-server architecture, which includes the creation of a PUF on a field-programmable gate array (FPGA) for experimental evaluation. Our implementation provides several insights concerning the adoption of PUF technology in security protocols.
- We provide empirical results that validate our use of a PUF to create a light-weight multifactor authentication system.

The rest of this paper is organized as follows. Section 2 details our threat model and deployment assumptions. Section 3 describes related work in the area of trusted computing, authentication, and access control. Section 4 provides an overview of how PUFs are created and controlled. In Section 5, we define our notion of *physically restricted access control*, specify protocols for enforcing this goal, and provide a formal analysis of our security guarantees. Section 6 provides details of our implementation, including our choices of cryptographic primitives for our protocol and our PUF FPGA implementation. Section 7 presents empirical results of our experiments. In Section 8, we discuss additional issues relevant to future implementations of our scheme. We then conclude in Section 9.

## 2. THREAT MODEL

In describing our threat model, we start with the central server  $S$ . We first note that the adversary’s goal is to gain access to sensitive data stored on  $S$ . We place no restrictions on what constitutes this data; we simply note that a server application running on  $S$  is responsible for the access control decisions. Next, we assume that  $S$  is trusted and secure. While this may seem like a strong assumption to make, we stress that it is the *data* stored on  $S$  that is important. That is, if an adversary can compromise  $S$ , there is no need to attack our protocols, as he has already “won.”

Regarding the client devices  $C$ , we assume that the organization has the authority to tightly control the software running on each device. While this is a daunting task for traditional computing, recall that we are also highly motivated by the concerns of embedded distributed applications. Embedded devices do not require the complex code base that exists in a traditional workstation; thus, satisfying this requirement is easier. Furthermore, our protocols will still apply in traditional schemes, too. Specifically, remote attestation techniques can be used to ensure that only known, trusted software is running.

Our main adversaries, then, are the users. We consider two classes of users as threats. First, client users have full access to the device, with the exception of installing software. That is, these users can read any data stored on the device. However, they cannot extract the data from memory to external storage. Also note, in the case of embedded systems, there might not actually be a user, as the devices may be executing autonomously. If there is a human user, he will have a password, and we assume it is protected.

The other class of users that pose a threat, whether malicious or not, are administrators. While administrators may have access to the data on  $S$  directly, our assumption is that the goal of a malicious administrator is to enable access to an untrusted device, thereby bypassing the physical restrictions. This adversary has access to all secret data stored on  $S$ .

Finally, we also consider network-based attackers, such as eavesdroppers. In all cases, we apply standard cryptographic assumptions. Specifically, we assume that adversaries are limited to probabilistic, polynomial-time attacks.

## 3. RELATED WORK

The literature of computer security contains a long history of identification schemes and authentication protocols [24, 23, 11, 12]. Modern research in this area has become more focused on addressing issues concerning digital identity management under specialized circumstances, such as internet banking [10], secure roaming with ID metasystems [20], digital identity in federation systems [5], authentication for location-based services [18], and location-based encryption [2]. These works rely on knowledge or possession of a secret, and do not bind the authentication request to a particular piece of hardware.

The origin of PUFs can be traced to attempts to identify hardware devices by mismatches in their behavior [22]. The use of PUFs for generating or storing cryptographic keys has been proposed in a number of works [31, 17, 16, 15, 14]. AEGIS [32, 33], a new design for a secure RISC processor, incorporates a PUF for cryptographic operations. Biometrics have also been used to generate secure keys [19]. We will contrast our approach with this scheme in Section 6.1. Our work contrasts with these, as we aim to integrate the unique PUF behavior directly into an authentication protocol, rather than simply providing secure key storage.

In a previous work, we presented a very rudimentary sketch of incorporating PUFs into an authentication system (reference omit-

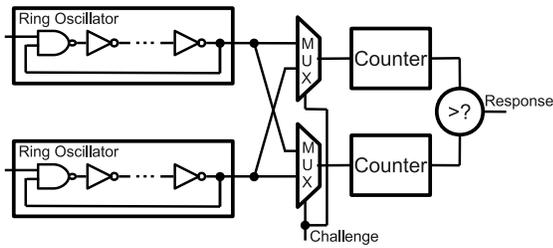


Figure 1: A sample 1-bit PUF based on ring oscillators

ted for purposes of anonymity). However, the focus of that paper was on joint installation of PUF challenges to combat insider threats. Additionally, that work did not present any formal protocol definition or implementation. In contrast, our current work presents substantial more significant results. We present formal definitions of our approach, protocols, and security proofs of our design. Also, our current work addresses the technical details involved with such an implementation, including the necessity of error-correcting codes, and presents empirical results of our prototype implementation.

Besides our previous work, [3] and [13] are perhaps the most similar to our current work. However, the former focuses on binding software in a virtual machine environment, whereas the latter focuses on authenticating banking transactions. Our protocols focus on light-weight multifactor authentication for distributed settings to bind remote file access to trusted systems.

Other types of trusted hardware exist for various purposes. TPMs can provide secure key storage and remote attestation [34, 4, 30, 28]. In many cases, the secure storage of TPMs can be used to bind authentication to a piece of hardware. However, we are interested in solutions for distributed computing that do not rely on TPMs, as TPMs may not be available for the devices used.

Finally, a new direction for hardware identification has emerged to identify unique characteristics of RFID devices [8, 7, 29]. These works are similar to previous work on PUFs, where they focus on identifying the device. These works do not propose new protocols that incorporate the unique behavior directly.

## 4. PUFs

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.

While there are several types of PUFs, in this work we focus on PUFs derived from ring oscillators (ROs). Figure 1 shows a sample 1-bit RO PUF. 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. In a 1-bit PUF, the output of two ROs pass through a pair of multiplexors (MUX) into a pair of counters that count the number of fluctuations between the 0 and 1 output. The PUF result is 1 if the counter on top holds a greater value, and

0 otherwise. The role of the challenge in a 1-bit RO PUF is to flip the MUX.

Clearly, it is not desirable to have such a one-to-one correspondence for larger PUFs. As such, for larger output bit strings it is better to have a pool of ROs, and randomly select pairs for comparison based on the challenge. In [33], the authors evaluate the entropy resulting from random pairings of ROs, and show that  $N$  ROs can be used to produce  $\log_2(N!)$  bits. For example, 35 ROs can be used to create 133 bits. Thus, a small number of ROs can be used to exhibit good random behavior. Another way to introduce entropy into the PUF behavior is to apply a cryptographic hash to the output. Given a strong hash function, changing a single bit of the PUF challenge, which yields a single flipped PUF bit, will produce a very different output.

The interesting properties of PUFs arise from the fact that it is virtually impossible for two ROs to operate at the same frequency. Specifically, miniscule variations in the wire width or length can cause one RO to oscillate at a faster speed than the other. As these variations are persistent, one of the oscillators will *consistently* be faster. Thus, the behavior of PUFs based on ROs depends on the physical instance of the device. Also, if the PUF is large enough, the behavior is unique. Furthermore, as these variations can be neither predicted nor controlled, they cannot be cloned.

With the exception of our implementation description in Section 6, we will assume an *idealized PUF* in our protocol design. That is, given a challenge-response pair  $\langle C_i, R_i \rangle$  and another challenge  $C_j \neq C_i$ , one cannot predict the value of  $R_j$ . Consequently, our results apply to any PUF that meets this ideal, rather than just RO-based PUFs.

## 5. PHYSICALLY RESTRICTED ACCESS CONTROL

In this section, we start by defining our notion of physically restricted access control. Next, we offer a high-level protocol and formal analysis for achieving this goal. We then present a more concrete example of this protocol that is derived from the Feige-Fiat-Shamir identification scheme.

We assume that the protected resources consist of files on a central server and subjects request access to these files remotely. For a file access request by a subject from a given device, the access control system checks whether the subject is allowed to access the file from the device; if this is the case, the server encrypts the file with a dynamically generated key and sends the resulting data to the device.

We thus assume an access control model based on a number of sets. Let  $\mathcal{S}$  denote the set of subjects,  $\mathcal{D}$  the set of trusted devices,  $\mathcal{F}$  the set of protected files, and  $\mathcal{R}$  the set of privileges. For simplicity, we assume  $\mathcal{R} = \{\text{read}, \text{write}\}$ . A permission can be written as the tuple  $\langle s, f, r \rangle$ , such that  $s \in \mathcal{S}$ ,  $f \in \mathcal{F}$ , and  $r \in \mathcal{R}$ . Thus  $\mathcal{P} \subseteq \mathcal{S} \times \mathcal{F} \times \mathcal{R}$  defines the set of authorized permissions subject to the physical restrictions. Let  $PUF_d : C \rightarrow R$  be the PUF for a trusted device  $d \in \mathcal{D}$ .

We define **physically restricted access control** to be the restriction of an access request  $\langle s, d, f, r \rangle$ , subject to the following conditions:

- The identity of  $s$  is authenticated.
- $\langle s, f, r \rangle \in \mathcal{P}$ .
- $d \in \mathcal{D}$ , and the authentication is performed implicitly by the ability of  $d$  to demonstrate a one-time proof of knowledge of  $PUF_d$ .

- A dynamic encryption key  $k_{PUF}$  based on the proof of  $PUF_d$  is used to bind the request to the device.

An important element of this definition is the notion of *hardware binding* of the cryptographic key. That is, the key  $k_{PUF}$  is generated dynamically and relies on the physical properties of the hardware itself (*i.e.*, the PUF). Consequently,  $k_{PUF}$  is never explicitly stored on the requesting device. This dynamic key generation is in contrast to traditional key management, in which keys are generated *a priori*. This approach simplifies the administration work, while reducing the threat of a rogue administrator transferring keys to an untrusted device.

One possible criticism to our definition is that it does not consider what happens to the contents of the file after decryption. That is, if the device  $d$  is malicious (or is infected with malicious software), it could simply broadcast the contents after decryption. We counter this objection by noting that remote attestation techniques could be applied to ensure that only trusted applications are running on the device. Hence, we assume either the device is free of malware, or the server can detect the malware and abort.

In addition to such software attacks, an attacker with physical access and sufficient technical skill could read the contents directly from the device's memory. However, such an attack exists regardless of the access control methodology applied. As such, we consider such threats beyond the scope of our work.

## 5.1 Protocols

Our protocols rely on a number of cryptographic primitives. Let  $H$  denote a collision-resistant hash function, while  $\text{Enc}_k(m)$  denotes the symmetric key encryption of a message  $m$  with the key  $k$ , using a cipher that is secure against *probabilistic polynomial time* (PPT) known ciphertext attacks. Define  $\text{Auth}(\cdot)$  to be a robust authentication scheme that is resilient against PPT adversaries.  $\text{Gen}(\cdot)$  denotes a pseudorandom key generator based on the provided seed value.

Let  $\text{Commit}(\cdot)$  denote a commitment scheme that ensures confidentiality against PPT adversaries.  $\text{Chal}(\cdot)$  and  $\text{Prove}(\cdot)$ , then, indicate a random challenge and the corresponding zero-knowledge proof of the secret value bound to the commitment. Furthermore, we assume that any PPT adversary  $\mathcal{A}$  has negligible probability of guessing  $\text{Prove}(\cdot)$  without access to the committed secret value. Assuming  $C$  denotes the PUF-enabled client (also called the device) and  $S$  indicates the server, the table in Table 1 gives the formal definition of our protocols.

Given these formalisms, we now explain the intuition behind each protocol. In  $\text{Request}(adm, m)$ , an administrator  $adm$  requests a set of  $m$  challenges to be used with a new (unspecified) device.<sup>1</sup>  $S$  authenticates  $adm$  and creates a database entry of the form  $\langle adm, n, C_1, \dots, C_m \rangle$ , binding those challenges and the nonce to that administrator. Hence, only that administrator is authorized to use that particular set of challenges. We use  $prms$  to indicate any parameters needed for the commitment and proof scheme. For instance, in our implementation  $prms$  consists of a modulus.

For  $\text{Enroll}(adm, pwd, C_1, \dots, C_m, prms)$ , we are assuming a trusted path from  $adm$  to  $C$ . That is, no eavesdropper learns the administrator's password, and all data are entered correctly. Based on this assumption,  $adm$  provides the inputs to  $C$ , which initiates an enrollment protocol that starts with authenticating  $adm$ .  $C$  uses a

<sup>1</sup>In general, we assume  $C_i \leftarrow \text{Gen}(1^n) \forall 1 \leq i \leq m$ ; that is, each challenge is the result of a pseudorandom generator with a security parameter  $1^n$ . However, in some applications, it may be desirable for  $S$  to define the challenges predictably. As such, we are intentionally vague on the selection of  $C_1, \dots, C_m$ .

pseudorandom generator to produce a one-time-use key  $otk$  derived from the administrator's password  $pwd$ , the nonce  $n$ , and the challenges.  $S$  can retrieve the nonce and challenges from its database, thus recreating the key on its end.  $C$  uses  $otk$  to encrypt a commitment of the PUF challenge-response pairs.  $S$  acknowledges receipt of the values with a hash of the commitment.

Finally,  $\text{Access}(user, file, action)$  defines the access request protocol. As before,  $S$  authenticates the user making the request, and selects a random set  $T$  of the challenges  $C_1, \dots, C_m$ . After receiving  $\text{Chal}(T)$ ,  $C$  executes the PUF to get the responses  $R_i$  for each  $C_i \in T$ . The corresponding zero-knowledge proof  $p \leftarrow \text{Prove}(T)$  is derived from these responses.  $S$  uses  $p$  and the user's password  $pwd$  as inputs to a pseudorandom generator to produce a one-time-use key  $k$ .  $S$  encrypts the file contents  $c$  with this key, returning the encrypted file to  $C$ . Hence, the intuition behind this protocol is that the file can only be decrypted by that user with that particular PUF-enabled device.

We note that there is one important consideration regarding our definition of  $\text{Access}(user, file, action)$ . Unlike the previous protocols, this protocol will be executed repeatedly. However, there are only  $2^m$  subsets of  $\mathcal{P}(C_1, \dots, C_m)$ . After all subsets are exhausted for a single user, the necessary proof will be reused. However, this repetition is acceptable, as the proof is never made public. Instead, the proof is used as an input to the key generation. Furthermore, assuming the nonce  $z$  is never repeated, the keys generated will always be different, even if  $p \leftarrow \text{Prove}(T)$  is reused.

In designing our protocols, we envisioned both traditional computing and embedded applications. In the embedded scenario, there may not be a human  $user$  making the request  $\text{Access}(user, file, action)$ . A straightforward variant of our protocol could accommodate this situation by eliminating  $\text{Auth}(user)$  from that protocol. Then,  $S$  must make the access control decision based on the device making the request, not the  $user$  doing so. Though this flexibility is a nice feature of our design, we will not investigate the security claims of this variant in this paper.

## 5.2 Security Analysis

Here, we present our formal analysis of the security properties of our protocols. We start with three lemmas, and complete our analysis with a theorem that our approach satisfies our definition of physically restricted access control.

### Lemma 1.

*A PPT adversary  $\mathcal{A}$  can enable an untrusted device with only negligible probability.*

**Proof:** Based on our assumption that  $\text{Auth}(\cdot)$  is resilient against PPT adversaries,  $S$  will abort the  $\text{Request}(\cdot)$  and  $\text{Enroll}(\cdot)$  protocols, except with negligible probability. Even with a transcript of  $\text{Request}(\cdot)$ ,  $\mathcal{A}$  must be able to forge the  $\text{Enc}_{otk}(\cdot)$  message to enable an untrusted device. However, with no knowledge of  $pwd$ , this feat is also infeasible, by our assumptions of  $\text{Enc}_k(\cdot)$ . Therefore,  $\mathcal{A}$  has only negligible probability of completing the  $\text{Enroll}(\cdot)$  protocol and enabling an untrusted device.  $\square$

### Lemma 2.

*An honest client  $C$  can validate its enrollment with the legitimate  $S$ , except with negligible probability.*

**Proof:** Similar to Lemma 1, a PPT adversary  $\mathcal{A}$  has negligible probability of forging  $H(\text{Commit}(\langle C_1, R_1 \rangle, \dots, \langle C_m, R_m \rangle))$ . Hence, if  $C$  receives such a hash, it has high assurance that the hash originated from the legitimate  $S$  and the enrollment succeeded.  $\square$

Request( $adm, m$ ) – Administrator $adm$ requests $m$ challenges to enable a new device.
– $S$ performs Auth( $adm$ ) – $S$ responds with $C_1, \dots, C_m$ , parameters $prms$ , and a nonce $n$
Enroll( $adm, pwd, C_1, \dots, C_m, prms$ ) – $C$ (after receiving data provided by $adm$ ) sends a commitment of the PUF to $S$ .
– $S$ performs Auth( $adm$ ) – $C$ generates $otk \leftarrow \text{Gen}(pwd, n, C_1, \dots, C_m)$ – $C$ provides $\text{Enc}_{otk}(\text{Commit}(\langle C_1, R_1 \rangle, \dots, \langle C_m, R_m \rangle))$ – $S$ responds with $\text{H}(\text{Commit}(\langle C_1, R_1 \rangle, \dots, \langle C_m, R_m \rangle))$
Access( $user, file, action$ ) – Subject $user$ requests $action$ for $file$ , which is encrypted with key $chal$ and transferred. If $action = read$ , $S$ sends the file. Otherwise, $C$ sends it.
– $S$ performs Auth( $user$ ) and issues $\text{Chal}(T)$ , where $T \subset \mathcal{P}(C_1, \dots, C_m)$ – $S$ responds with a nonce $z$ – $S$ verifies that $user$ is permitted to perform $action$ on $file$ – Generate and transfer $\text{Enc}_{chal}(c)$ , where $p \leftarrow \text{Prove}(T)$ and $chal \leftarrow \text{Gen}(p, z, pwd)$

**Table 1: Protocols for enforcing physically restricted access control**

### Lemma 3.

A PPT adversary  $\mathcal{A}$  with transcripts of Request( $\cdot$ ) and Enroll( $\cdot$ ) can model the PUF with only negligible probability.

**Proof:** In order for  $\mathcal{A}$  to learn the commitments of the PUF behavior,  $\mathcal{A}$  must either decrypt  $\text{Enc}_{otk}(\text{Commit}(\langle C_1, R_1 \rangle, \dots, \langle C_m, R_m \rangle))$  or find a preimage of  $\text{H}(\text{Commit}(\langle C_1, R_1 \rangle, \dots, \langle C_m, R_m \rangle))$ . However, based on our assumptions regarding  $\text{Enc}_k(\cdot)$  and  $\text{H}(\cdot)$ , both actions are infeasible. Thus, these protocols do not leak enough information for a PPT adversary  $\mathcal{A}$  to model the PUF.  $\square$

Informally, these lemmas demonstrate that the Request( $\cdot$ ) and Enroll( $\cdot$ ) protocols guarantee integrity and confidentiality against PPT adversaries. That is, by viewing a transcript of both protocols,  $\mathcal{A}$  fails to learn the administrator’s  $pwd$  or the PUF challenge-response pairs. Furthermore, any tampering by  $\mathcal{A}$  will be detected by either  $S$  or  $C$ . Also,  $\mathcal{A}$  cannot launch a man-in-the-middle attack against Enroll( $\cdot$ ), as doing so requires knowledge  $pwd$ . Applying these lemmas, we propose the following theorem.

### Theorem 1.

The Access( $\cdot$ ) protocol enforces physically restricted access control under the PPT adversarial model.

**Proof:** By Lemma 1, we are guaranteed that only trusted devices will be able to produce  $p \leftarrow \text{Prove}(T)$ . Lemma 2 ensures that trusted devices receive confirmation if their enrollment is successful; as such, if the confirmation is not received, proper mitigation can be performed. By Lemma 3, we are guaranteed that PPT adversaries cannot possess a model of the PUF behavior by observing a transcript of the Request( $\cdot$ ) and Enroll( $\cdot$ ) protocols. We explicitly model the authentication of  $user$ , check that  $user$  is authorized to perform  $action$  on  $file$ , and the device is implicitly authenticated by generating a one-time proof of knowledge of the PUF behavior. Furthermore, the one-time key  $chal \leftarrow \text{Gen}(p, z, pwd)$  exists only at run-time, is never transmitted, is bound to the hardware of the requesting (trusted) device (by the use of the PUF), and is used to encrypt data transferred between  $C$  and  $S$ . The probability of a PPT adversary generating  $chal$  is negligible, so the encryption successfully enforces the access control policy. Therefore, by definition, the Access( $\cdot$ ) protocol enforces physically restricted access control under the PPT adversarial model.  $\square$

## 6. IMPLEMENTATION

In this section, we describe our implementation of a PUF-based access control mechanism based on our protocols described above. We start by describing our protocol instantiation and our implemen-

tation of a PUF using ring oscillators, which is the same method used in [32]. We also describe the use of Reed-Solomon codes to ensure the PUF produces a consistent result that can be used for authentication, and detail our minimal storage requirements.

### 6.1 Protocol Instantiation

The underlying premise of our protocol instantiation is the Feige-Fiat-Shamir identification scheme. Our choice of hash function was SHA-1, although a better choice would be SHA-256, which offers more protection against preimage attacks and is collision-resistant. Our choice of symmetric key cryptography was AES which also provides the security against PPT adversaries that we require.

Our Auth( $\cdot$ ) primitive uses the hash function and a nonce  $n$  in a challenge-response protocol. Specifically,  $S$  generates  $n$ , and the user must respond with  $\text{H}(\text{H}(pwd), n)$ . Note that both hashes are necessary, as our implementation of  $S$  protects the secrecy of user passwords by storing  $\text{H}(pwd)$ , not the passwords themselves. Furthermore, as the response requires knowledge of both  $n$  and the password (in the form of  $\text{H}(pwd)$ ), this challenge-response pair preserves the secrecy of  $pwd$  from PPT adversaries. Figure 2(a) shows our implementation of Request( $adm, m$ ), in which an administrator  $A$  requests a new set of challenges from the server  $S$ . The parameter  $N$  returned in step 4 is used as a modulus in the other protocols.

Our Enroll( $adm, pwd, C_1, \dots, C_m, prms$ ) implementation is shown in Figure 2(b). Our Commit( $\cdot$ ) primitive consists of the pairs  $(C_1, R_1^2), \dots, (C_m, R_m^2)$ , where the multiplication is modulus  $N$ . The security of this commitment relies on the intractability of computing  $R_i$  by observing  $R_i^2 \pmod{N}$ . That is, even if a PPT adversary gains access to the committed values stored on  $S$ , he can compute the modular square roots with only negligible probability, and the confidentiality of the PUF is assured. As we will explain in Subsection 6.4, we used the `mcrypt` utility to generate the cryptographic keys, thus providing the functionality of the Gen( $\cdot$ ) primitive.

Our instantiation of Access( $user, file, action$ ) is shown in Figure 2(c). As we mentioned previously, our choice of Chal( $\cdot$ ) and Prove( $\cdot$ ) is based on the Feige-Fiat-Shamir identification scheme. The first step of this scheme is for the prover ( $C$ ) to generate a random  $r$  and send  $x \equiv \pm r^2 \pmod{N}$ .<sup>2</sup> The user is then authenticated using a nonce and a cryptographic hash. Given the challenge set  $T \subset \mathcal{P}(C_1, \dots, C_m)$  (where  $\mathcal{P}$  denotes the power set),  $C$  executes the PUF for each  $C_i \in T$ . That is,  $C$  computes  $y \equiv r \cdot \prod R_i^{p_i} \pmod{N}$ , where  $p_i = 1$  if  $C_i \in T$  and  $p_i = 0$

<sup>2</sup>Randomly flipping the sign of  $r^2 \pmod{N}$  ensures that the scheme is a zero-knowledge proof of knowledge.

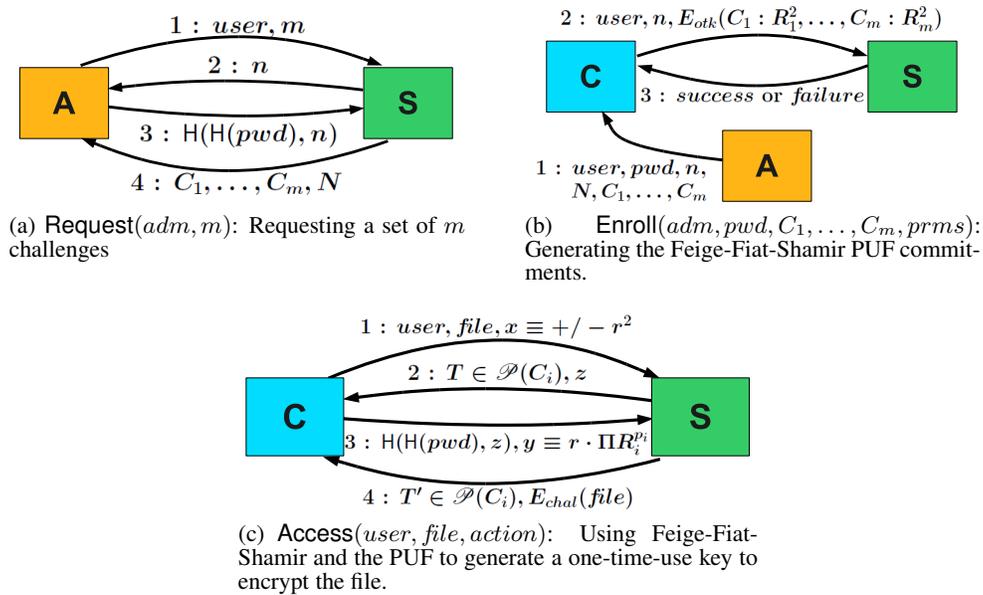


Figure 2: Physically restricted access control protocols. All multiplications are modulo  $N$ .

otherwise. Thus, the proof  $p \leftarrow \text{Prove}(T)$  is the value  $\pm y^2 \pmod{N}$ . As both parties also know  $H(pwd)$  and the nonce  $z$ , and they can compute  $\pm y^2 \pmod{N}$ , they can use the proof to generate  $chal \leftarrow \text{Gen}(p, z, pwd)$  as required by the protocol.

There is an important subtlety here that should be noted. Under the traditional Feige-Fiat-Shamir scheme, the prover sends  $y$  and the verifier must compare both  $y^2 \pmod{N}$  and  $-y^2 \pmod{N}$  with the product of  $x$  and the committed values. That is, it would seem that  $C$  and  $S$  would have to attempt the encryption and/or decryption twice. However, this is not the case.  $S$  always uses  $x \cdot \prod R_i^{p_i}$  (which includes the correct sign). As the decision of whether or not to flip the sign of  $x$  was made by  $C$ ,  $C$  clearly knows whether the proof should be  $y^2 \pmod{N}$  or  $-y^2 \pmod{N}$ . Hence, the encryption and decryption only need to be attempted once by each party.

In addition, readers who are familiar with existing work in generating cryptographic keys from biometrics [19] may object to our use of the responses as the secrets. In that work, the authors create a secure key  $\mathcal{K}$  and compute  $\Theta_{lock} = \mathcal{K} \oplus \Theta_{ref}$ , where  $\Theta_{ref}$  denotes the reference biometric sample. To authenticate a sample  $\Theta_{sam}$  at a later point, the system applies the bit mask  $\Theta_{lock}$  in an attempt to recover the key  $\mathcal{K}$ .

In our approach, this bit mask is unnecessary for two reasons. First, unlike biometric data, the PUF responses exist only at runtime and are never made public. In contrast, biometric data, such as fingerprints, are always present and can be harvested. Thus, PUF responses are more private and, consequently, more protected. Second, revoking a biometric is impossible; however, it must be possible to revoke the associated key. The bit mask makes this possible. In our scheme, though, revocation of a PUF response  $R_i$  is simple:  $S$  stops using the associated challenge  $C_i$ . Hence, applying the bit mask to the PUF response is unnecessary for our scheme.

## 6.2 PUF Creation

We used the Xilinx Spartan-3 FPGA to implement a PUF. To simplify the circuitry, we created independent pairs of ROs, each forming a 1-bit PUF. To ensure that we could count a high number of oscillations, we implemented a 64-bit counter to receive the data

from each multiplexor. Each oscillator consisted of a series of nine inverter gates. Our experiments with fewer gates resulted in the oscillator running at too high of a frequency, but nine gates offered good, consistent behavior.

We controlled the PUF execution time by incrementing a small counter until it overflowed. The Spartan-3 uses a 50 MHz clock, so a 16-bit counter overflows in approximately 1 ms. We also increased the counter size to 20 bits, which required 21 ms to overflow. We did not notice any observable difference in the consistency; hence, a 16-bit counter offers sufficient time for the oscillators to demonstrate quantifiably different behavior.

Our design is based on a 128-bit PUF. However, in our experiments, we needed to create a state machine to write the PUF result out to a serial port. The extra space for the state machine would not fit on the Spartan-3. As such, we reduced the PUF size to 64 bits for experimental evaluation. In future designs, all work will be performed on the device itself, the state machine will not be needed, and accommodating 128-bit PUFs (and larger) will certainly be feasible.

From the perspective of space on the device, the limiting factor is the usage of the look-up tables (LUTs). Implementing a 128-bit PUF on the Spartan-3 occupies 39% of the available input LUTs and 78% of slices. However, as more ROs are added, the number of slices grows only slightly, while the usage of the LUTs increases more quickly. Implementing two independent 128-bit PUFs on the same device would occupy 78% of the LUTs and 99% of slices. Note, though, that these numbers are based on our simplistic PUF design, which consists of 128 pairs of independent 1-bit ROs. More advanced designs [33] select random pairs from a pool of ROs; in such an approach, a 128-bit output can be produced from 35 ROs, whereas our approach would use 256 (128 pairs).

By implementing the full PUF as independent 1-bit PUFs, there is a direct correlation between each bit of the challenge and each bit of the response. That is, flipping only a single bit of input would result in only a single bit difference in the output. To counteract this correlation, we take a hash of the PUF output. As a result of the properties of cryptographic hash functions, a single bit difference in the PUF output will produce a very different hash result. This

hash step prevents an attacker from using the one-to-one mapping to model the PUF.

### 6.3 Error Correction

PUFs are designed to be generally non-deterministic in their behavior. The physical properties of the device itself resolves this non-determinism to create a consistent and predictable challenge-response pattern. However, variations in the response are inevitable. For instance, if two ring oscillators operate at nearly identical frequencies, the PUF may alternate between identifying each as the “faster” oscillator. Reed-Solomon codes [26] correct these variations up to a pre-defined threshold.

Reed-Solomon codes are linear block codes that append blocks of data with parity bits that can be used to detect and correct errors in the block. To guarantee that we can correct up to 16 bits of output for a 128-bit PUF, we use a RS(255,223) code. Note that this code operates on an array of *bytes*, rather than bits. To accommodate this, we encode each PUF output bit into a separate byte. Alternatively, we could have compacted eight bits at a time into a single byte for a more compact representation. In fact, doing so is necessary for implementations that use larger sizes of PUF output. For our current work, though, we find this encoding to be acceptable, even if it is not optimal.

RS(255,223) reads a block of 223 input symbols and can correct up to 16 errors. After converting the PUF output to a string of bytes, we pad the end of the string with 0s. The encoding produces a *syndrome* of 32 bytes that must be stored. When the PUF is executed at a later point, the response is again converted to a string of bytes and padded, and these 32 bytes are appended. The array of bytes is then decoded, correcting up to 16 errors introduced by the noisy output of the PUF.

While Reed-Solomon codes can correct errors in a data block, they operate under the assumption that the original data is correct. In the case of PUFs, it is also possible that the original data varies from the normal behavior observed at later times. To counteract this initial bias, during the enrollment process, we execute the PUF three times, not once. For each bit, we do a simple majority vote. That is, the “official” PUF result is the result of the consensus of the three executions.

### 6.4 Client-Server Implementation

We implemented our protocols as a custom client-server prototype. Both applications use a custom-built package for performing arbitrary-length arithmetic operations for large numbers. All hash operations use the SHA-1 implementation by Devine [9]. We incorporated the Reed-Solomon code library created by Rockliff [27]. Recall that, in our protocols, we use symmetric key encryption in a number of steps; the symmetric keys are generated from a shared secret. In all cases, we wrote the secret to a file, used the Linux utility `mcrypt` (which reads the file and generates a strong key from the data), and immediately destroyed the file using `shred`. The cryptographic algorithm used was 128-bit AES (Rijndael). To minimize the possibility of leaking the key by writing the shared secret to a file, we used `setuid` to run server under a dedicated uid, and restricted read access to the file before writing the secret.

### 6.5 Storage Requirements

The storage requirements of our solution for both  $C$  and  $S$  are minimal.  $C$  must store  $N$ , the challenges  $C_i$ , and an error-correcting syndrome for each challenge. As we detailed above,  $N$  and  $C_i$  are each 128 bits, or 16 bytes in length. Each syndrome (one per challenge) is 32 bytes in length. Thus, the total storage for  $C$  in our

prototype is  $48m + 16$  bytes. For 16 challenges, then, the storage requirement is under 1 KB.

$S$  also must store a minimal amount of data.  $S$  stores  $N$  and the  $R_i^2 \pmod{N}$  commitments, each of which are 128 bits (16 bytes) in size. In addition,  $S$  stores a hash of each user’s password. If SHA-1 is used, that hash is 20 bytes. If  $a$  denotes the number of devices enabled and  $b$  denotes the number of authorized users, the total storage requirement for our system is  $(16m+16)a+20b$  bytes of data. *E.g.*, given 100 users,  $S$  can enable 1000 devices with 16 challenges each for less than 268 KB of storage.

## 7. EXPERIMENTAL EVALUATION

We now present the experimental evaluation of our prototype. Our evaluation goals focused on two areas. First, we strove to demonstrate that RO-based PUFs are both non-deterministic and consistent. That is, different physical instantiations of the same PUF design produce different behavior, but repeating the PUF execution on the same input and hardware produce results that can be reliably quantified as the same binary string. Our second area of evaluation was on the performance of our client-server prototype. In that portion, we show that our design offers better performance than using traditional PKI to distribute symmetric encryption keys.

The output from the PUF, implemented using a Xilinx Spartan-3 FPGA, is transferred to a client application via serial cable, although in deployed settings all operations would occur on the same device. All client and server operations were executed on a system with a 2.26GHz Intel® Core™ 2 Duo CPU with 3GB of 667MHz memory. The OS used was Ubuntu 9.04, with version 2.6.28-15 of the Linux kernel.

### 7.1 PUF Consistency

As noted in Section 6, we implemented a 64-bit PUF and wrote the serialized output to a workstation via cable. In our experiments, we observed an average of 0.2 bits that differed from the “official” PUF result. The maximum difference that we observed was 5 bits. Clearly, the use of Reed-Solomon codes that can correct up to 16 error bits at each iteration will be able to provide consistent output from the PUF, even if we double the size of the PUF to 128 bits. Furthermore, note that changes in environmental conditions, such as different temperatures, will affect the absolute speeds at which the ROs oscillate. However, the PUF result is based on the *relative* speeds; that is, increasing the temperature will slow both ROs in a pair down, but is unlikely to change which of the two oscillates faster. Consequently, the PUF shows very consistent behavior that can be used to build a reliable authentication mechanism.

### 7.2 Client/Server Performance

To evaluate the performance of our client and server implementations, we executed a series of automated file requests, given several different files sizes. In these experiments, we emulated the PUF in software. As noted in Section 6.2, we can control the PUF execution time; overflowing a 16-bit counter adds only 1 ms to the client computation time. Figures 3 and 4 report the amount of time for computing key portions of the Access protocol for some of the file sizes that we measured.

In these figures, “Generate Proof” (shown in blue) refers to the time to authenticate the user by generating or checking the hash  $H(H(pwd), z)$  and the proof  $y$  sent in step 3. “Generate Key” (shown in green) refers to the amount of time required to create the 128-bit AES key needed to encrypt or decrypt the file,  $E_{chal}(file)$ . The AES computation is shown in orange.

Figures 3 and 4 are shown on both a (truncated) linear scale and a logarithmic scale. The key observation of these figures is that the

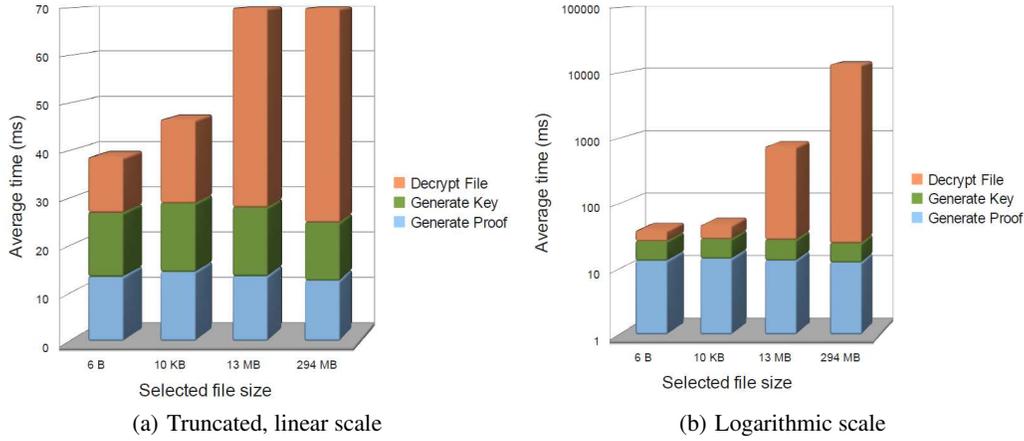


Figure 3: Average client-side computation time for steps 3 and 4 of the Access protocol.

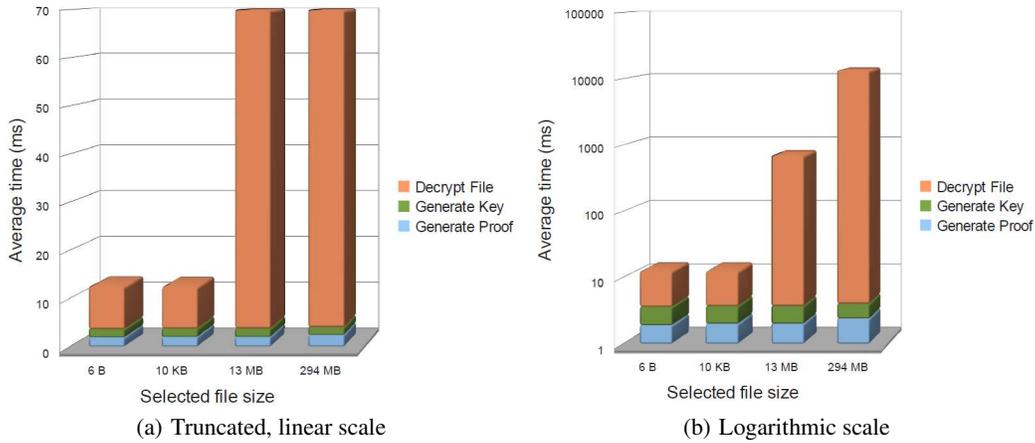


Figure 4: Average server-side computation time for steps 3 and 4 of the Access protocol.

two primary functions of our protocol, plotted as “Generate Key” and “Generate Proof,” are fairly constant and minimal. The client side operations take approximately 14 ms on average, which is the same length of time as decrypting a 6-byte piece of data with AES (12 ms on average). The server burden is even less, requiring approximately 2 ms for each protocol stage and 9 ms to encrypt the file. As the file size increases, the AES encryption clearly becomes the limiting factor, as it increases approximately linearly with the file size, while our protocol overhead remains constant.

Comparing the performance of our approach with traditional PKI (specifically, RSA) required addressing a number of factors. First, the intractability assumption behind our approach (as described in the next section) states that finding the modular square root is at least as hard as factoring the product of primes, *assuming the product and the modular square are the same size*. That is, computing  $R_i$  from a 128-bit  $R_i^2$  is only as difficult as breaking a 128-bit RSA key, which is quite a weak claim. Thus, we needed to increase the size of the PUF output. Note, though, that the PUF execution time does not change. The only additional performance overhead is the extra time required to do the modular multiplication on larger numbers.

The other disparity between our approach and RSA is that the result of an RSA decryption would give you the key itself. In our approach, we would be left with a 1024-, 2048-, or 4096-bit value that

would have to be converted into an AES key. However, based on our experiments with `mencrypt`, we observed only negligible overhead to convert this PUF output value into a key. Thus, this extra work had no measurable impact on our performance.

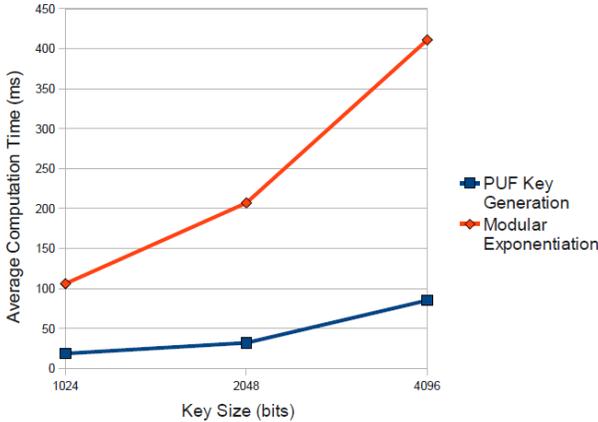
Figure 5 shows the difference in performance between our PUF-based key generation and using RSA to encrypt an AES key. The RSA modular exponentiation requires approximately four times the computation time as our client-side PUF-based key generation. Thus, our approach offers a clear performance advantage, which may be very beneficial for low-power embedded devices.

## 8. DISCUSSION

We start this section with a brief discussion on PUF and RSA key sizes. We then focus on possible attack models for our design.

### 8.1 On Key Sizes

In the previous section, we showed the performance difference between our 128-bit PUF-based client-server architecture and various sizes of RSA keys. However, comparing the security guarantees of our system with the use of PKI to distribute symmetric keys is somewhat challenging. Revealing  $R_i^2$  while assuming  $R_i$  to be secure relies on the assumption that computing modular square roots is intractable. [25] shows that this computation is at least as difficult as factoring the product of primes, *provided the num-*



**Figure 5: Large PUF computation compared with RSA-based modular exponentiation**

bers are all large. Intuitively, though, computing a 128-bit modular square root is only as hard as factoring a 128-bit RSA key, which is quite a weak claim. We counter this criticism of our design with the following justifications.

First, attacking the  $R_i$  values in this manner can only occur at  $S$ . That is, the  $R_i^2$  values are never transmitted in the clear where an attacker can eavesdrop. In RSA, though, public keys are used to encrypt the symmetric keys before transmitting them across the network. Transmitting keys in this manner creates an attack surface that our approach avoids.

Second, the PUF could be repeatedly polled to produce a larger output bit string. That is, appending 8 responses for a 128-bit PUF will create a 1024-bit bit string. Additionally, we showed that increasing the size yields a minimal performance cost when compared with common RSA key sizes. Consequently, we do not consider criticisms based on the key size to detract from the soundness of our overall design.

## 8.2 Additional Threats & Attacks

In Section 5.2, we provided a formal analysis of our protocol. Here, we expand on this analysis with an informal discussion of the remaining threats to our design. First, recall that our protocol is built on the assumption that  $C$  is a trusted device. As such, we do not consider attacks in which  $C$  leaks secure data received through a legitimate access request. The presence of malware on  $C$  makes this a very realistic concern. However, we consider this threat beyond the scope of our work, and focus on what can be accomplished under the assumption that the  $C$  is trusted.

A common flaw in authentication protocols is vulnerability to a replay attack. Consider a PPT adversary  $\mathcal{A}$  with a transcript of of  $\text{Access}(user, file, action)$ , as shown in Figure 2(c). If either  $z$  or  $T$  were different, the replay attempt would fail. Additionally, even if both  $z$  and  $T$  are the same,  $\mathcal{A}$  would learn nothing new. That is, under the PPT assumption,  $\mathcal{A}$  cannot decrypt  $E_{chal}(file)$ . The only threat in this scenario would be if the session involved *uploading* the file from  $C$  to  $S$ . In this case,  $\mathcal{A}$  could force  $S$  to revert the status of  $file$  to an earlier version. However, this can only happen if both  $z$  and  $T$  are identical. Assuming a large range of values for these variables, this attack can succeed with only negligible probability.

Now consider a stronger adversary  $\mathcal{A}$  that has learned the pairs  $(C_i, R_i^2)$  for a particular device. Under the PPT model, such an

adversary can only have learned these values by successfully attacking  $S$ . Clearly, if  $\mathcal{A}$  can bypass  $S$ 's protection of the pairs, he can also directly access all of the files on the system. Hence, the only remaining motivation of such an attacker is to try to model the PUF by learning the PUF responses.

The defenses against such an adversary rely on a number of factors. First, even if we set aside the PPT model and assume that the adversary has somehow learned the key used to encrypt  $E_{chal}(file)$  and the inputs to  $\text{Gen}(p, z, pwd)$ . Note that this  $p$  is exactly the proof generated in the Feige-Fiat-Shamir identification scheme, which is known to be zero-knowledge. Hence, observing additional sessions provides no new information regarding the values of  $R_i$ .

Thus,  $\mathcal{A}$  can only model the PUF by computing the modular square roots. Returning to the PPT model, such an attack can succeed with only negligible probability, as computing modular square roots is at least as difficult as factoring a large product of primes for composite values of  $N$  [25]. Admittedly, in our prototype, we used only 128-bit values (which is quite weak), but we demonstrated that it would be straightforward to increase the PUF output to larger sizes with minimal overhead. Hence, a PPT adversary could not model the PUF, even with possession of the pairs  $(C_i, R_i^2)$ .

Finally, consider the case of a malicious administrator. Insider threats are very difficult to prevent in general, as these attackers have been granted permissions because they were deemed trustworthy. In our approach, there is no inherent mechanism for preventing a malicious administrator from enabling untrusted devices. One simple defense would be to apply separation-of-duty, thus requiring multiple administrators to input the same challenges to each device.<sup>3</sup> Another approach would be to require a supervisor to approve the enrollment request. Incorporating such defense-in-depth techniques would strengthen our scheme against these threats.

## 9. CONCLUSIONS

In this work, we have proposed a novel mechanism that uses PUFs to bind an access request to a trusted physical device. In contrast to previous work, we do not use the PUF to generate or store a cryptographic key. Rather, we incorporate the PUF challenge-response mechanism directly into our authentication and access request protocols. Furthermore, our approach avoids expensive computation, such as the modular exponentiation used in public key cryptography. As a result, our PUF-based mechanism can be used in settings where PKI or TPMs are either not available or require too much performance overhead. We have presented the details of our implementation. Our empirical results show that PUFs can be used to create a light-weight multifactor authentication that successfully binds an access request to a physical device.

## 10. ACKNOWLEDGEMENTS

The work reported in this paper has been partially funded by Sypris Electronics.

## 11. REFERENCES

- [1] S. Aich, S. Sural, and A. K. Majumdar. STARBAC: Spatiotemporal role based access control. In *OTM Conferences, 2007*.
- [2] J. Al-Muhtadi, R. Hill, R. Campbell, and M. D. Mickunas. Context and location-aware encryption for pervasive computing environments. In *Proceedings of the 4th IEEE Conference on Security in Pervasive Computing and Communications, March 2006*.

<sup>3</sup>Of course, this does nothing against colluding administrators!

- [3] 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.
- [4] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vtpm: virtualizing the trusted platform module. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.
- [5] A. Bhargav-Spantzel, A. C. Squicciarini, and E. Bertino. Establishing and protecting digital identity in federation systems. In *Proceedings of the 2005 ACM Workshop on Digital Identity Management*, pages 11–19. ACM, 2005.
- [6] M. L. Damiani, E. Bertino, B. Catania, and P. Perlasca. GEO-RBAC: A spatially aware RBAC. *ACM Transactions on Information Systems and Security*, 2006.
- [7] B. Danev, T. S. Heydt-Benjamin, and S. Čapkun. Physical-layer identification of RFID devices. In *Proceedings of the USENIX Security Symposium*, 2009.
- [8] S. Devadas, E. Suh, S. Paral, R. Sowell, T. Ziola, and V. Khandelwal. Design and implementation of PUF-based “unclonable” RFID ICs for anti-counterfeiting and security applications. In *2008 IEEE International Conference on RFID*, pages 58–64, 2008.
- [9] C. Devine. FIPS-180-1 compliant SHA-1 implementation. <http://csourcesearch.net/c/fid1A3BFA49A2F9E1FFB3147B7238E287C22E7ED0A3.aspx>, 2006.
- [10] Federal Financial Institutions Examination Council. *Authentication in an Internet Banking Environment*, October 2005.
- [11] U. Feige, A. Fiat, and A. Shamir. Zero knowledge proofs of identity. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 210–217, 1987.
- [12] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Proceedings on Advances in Cryptology (CRYPTO '86)*, pages 186–194. Springer-Verlag, 1987.
- [13] K. B. Frikken, M. Blanton, and M. J. Atallah. Robust authentication using physically unclonable functions. In *Information Security Conference (ISC)*, September 2009.
- [14] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas. Controlled physical random functions. In *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC)*, 2002.
- [15] 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.
- [16] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls. FPGA intrinsic PUFs and their use for IP protection. In *Proceedings of the 9th Cryptographic Hardware and Embedded Systems Workshop (CHES)*, pages 63–80, 2007.
- [17] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls. Physical unclonable functions and public-key crypto for FPGA IP protection. In *International Conference on Field Programmable Logic and Applications*, pages 189–195, 2007.
- [18] K. Han and K. Kim. Enhancing privacy and authentication for location based service using trusted authority. In *2nd Joint Workshop on Information Security*. Information and Communication System Security, 2007.
- [19] F. Hao, R. Anderson, and J. Daugman. Combining crypto with biometrics effectively. *IEEE Trans. Comput.*, 55(9):1081–1088, 2006.
- [20] L. N. Hoang, P. Laitinen, and N. Asokan. Secure roaming with identity metasystems. In *IDtrust '08*. ACM, March 2008.
- [21] D. Kulkarni and A. Tripathi. Context-aware role-based access control in pervasive computing systems. In *Proceedings of the 14th Symposium on Access Control Models and Technologies (SACMAT)*, 2008.
- [22] K. Lofstrom, W. Daasch, and D. Taylor. IC identification circuit using device mismatch. In *Solid-State Circuits Conference, 2000. Digest of Technical Papers. ISSCC. 2000 IEEE International*, pages 372–373, 2000.
- [23] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer. Kerberos authentication and authorization system. In *Project Athena Technical Plan*, 1987.
- [24] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.
- [25] M. O. Rabin. Digitalized signatures and public-key functions as intractable as factorization. Technical Report MIT/LCS/TR-212, MIT Laboratory for Computer Science, January 1979.
- [26] 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](http://www.cs.cmu.edu/afs/cs.cmu.edu/project/pscico-guyb/realworld/www/reedsolomon/reed_solomon_codes.html), 1998.
- [27] S. Rockliff. The error correcting codes (ecc) page. <http://www.eccpage.com/>, 2008.
- [28] R. Sailer, T. Jaeger, X. Zhang, and L. van Doorn. Attestation-based policy enforcement for remote access. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*, pages 308–317. ACM Press, 2004.
- [29] 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.
- [30] D. Schellekens, B. Wyseur, and B. Preneel. Remote attestation on legacy operating systems with trusted platform modules. In *Science of Computer Programming*, pages 13–22, 2008.
- [31] G. E. Suh and S. Devadas. Physical unclonable functions for device authentication and secret key generation. In *Proceedings of the 44th IEEE Design Automation Conference (DAC)*, pages 9–14. IEEE Press, 2007.
- [32] G. E. Suh, C. W. O’Donnell, and S. Devadas. AEGIS: A single-chip secure processor. In *Elsevier Information Security Technical Report*, volume 10, pages 63–73, 2005.
- [33] G. E. Suh, C. W. O’Donnell, and S. Devadas. Aegis: A single-chip secure processor. *IEEE Design and Test of Computers*, 24(6):570–580, 2007.
- [34] Trusted Computing Group. Trusted Platform Module Main Specification. <http://www.trustedcomputinggroup.org/>, October 2003.
- [35] Verizon RISK Team. 2010 data breach investigations report. Technical report, 2010.