Controlled Physical Random Functions and Applications

BLAISE GASSEND, MARTEN VAN DIJK, DWAINE CLARKE, EMINA TORLAK, and SRINIVAS DEVADAS
Massachusetts Institute of Technology
and
PIM TUYLS
Philips Research

The cryptographic protocols that we use in everyday life rely on the secure storage of keys in consumer devices. Protecting these keys from invasive attackers, who open a device to steal its key, is a challenging problem. We propose controlled physical random functions (CPUFs) as an alternative to storing keys and describe the core protocols that are needed to use CPUFs. A physical random functions (PUF) is a physical system with an input and output. The functional relationship between input and output looks like that of a random function. The particular relationship is unique to a specific instance of a PUF, hence, one needs access to a particular PUF instance to evaluate the function it embodies. The cryptographic applications of a PUF are quite limited unless the PUF is combined with an algorithm that limits the ways in which the PUF can be evaluated; this is a CPUF. A major difficulty in using CPUFs is that you can only know a small set of outputs of the PUF—the unknown outputs being unrelated to the known ones. We present protocols that get around this difficulty and allow a chain of trust to be established between the CPUF manufacturer and a party that wishes to interact securely with the PUF device. We also present some elementary applications, such as certified execution.

Categories and Subject Descriptors: K.6.5 [**General**]: Security and Protection—*Physical Security*

General Terms: Security, Theory

Additional Key Words and Phrases: Certified execution, physical security, physical random function, physical unclonable function, trusted computing

ACM Reference Format:

Gassend, B., van Dijk, M., Clarke, D., Torlak, E., Devadas, S., and Tuyls, P. 2008. Controlled physical random functions and applications. ACM Trans. Inform. Syst. Secur. 10, 4, Article 15 (January 2008), 22 pages. DOI = 10.1145/1284680.1284683 http://doi.acm.org/10.1145/1284680.1284683

Authors' address: Blaise Gassend, Marten van Dijk, Dwaine Clarke, Emina Torlak, and Srinivas Devadas, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139; email: gassend@alum.mit.edu; Pim Tuyls, Philips Research, ISS WB 6038, Prof Holstlaan 4, Eindhoven.

1. INTRODUCTION

Typically, cryptography is used to secure communication between two parties connected by an untrusted network. In such communication, each party has privately stored key information which allows it to encrypt, decrypt, and authenticate the communication. It is implicitly assumed that each party is capable of securing its private information. This assumption is reasonable when a party is a military installation, or even a person, but breaks down completely for low-cost consumer devices. Once a secret key is compromised, eavesdropping and impersonation attacks become possible. In a world where portable devices need to authenticate credit card transactions and prevent copyright circumvention, protecting the keys in the devices that surround us is of utmost importance.

To be successful, a key protection scheme has to protect keys from application programming (API) attacks in which the device's API is tricked into releasing trusted information, from noninvasive attacks in which the key is deduced from unintended signals emanating from the device, and from invasive attacks in which the attacker opens the device to find the key. All these types of attacks have been demonstrated in real systems [Kocher et al. 1999; Anderson and Kuhn 1996, 1997; Gutman 1996].

Focusing on the problem of invasive attacks, it is apparent that once a device has been opened, the large difference in state between a 0 and a 1 makes it relatively easy to read out the device's digitally stored secrets. Traditionally, such attacks are avoided by detecting intrusion and erasing the key memory when an intrusion is detected [Smith and Weingart 1999]. However, tamper-sensing environments are expensive to produce and, as long as a key is being protected, the intrusion sensors need to be powered, further increasing costs.

Since it is the digital nature of the secret key material that makes it easy to extract invasively, we can try to use information of a continuous nature instead. For example, by measuring a complex physical system, and performing suitable processing, a key can be generated [Gassend 2003; Suh et al. 2005; Skoric et al. 2005]. The invasive adversary now has to study a complex physical system, measure it, and simulate it precisely enough to determine the device's key. With careful design, the complex physical system can be fabricated such that an invasive adversary, who wants to measure it, has to destroy it in the process. Thus unless the adversary successfully models or clones the physical system, his tampering will be noticed.

These physically obfuscated keys seem to increase the difficulty of an attack, but they still have a single digital point of failure. When the device is in use, the single physically obfuscated master key is present on it in digital form. If an adversary can get that key he has totally broken the device's security. Going one step farther, we get to physical random functions: instead of being used to generate the same key every time, the complex physical system is parameterizable. For each input to the physical system, a different key is produced. Thus the complexity of the physical system is exploited to the utmost.

However, by getting rid of the single digital point of failure, we have produced a device that has a family of secrets that are not computationally related to each other. In public key cryptography, publishing the public key of a device is sufficient to allow anybody to interact securely with that device. With physical random functions, if the multiplicity of generated keys is to be used, then there is no direct equivalent to publishing a public key. Our main contribution in this paper is to show how to build a key management infrastructure that exploits all the key material provided by a physical random function. We also present some basic applications, like certified execution. Because there is no algorithmic way to tie together all the keys produced by a given device, the device will have to take an active part in protocols like certificate verification, that would not usually need any device involvement. This limitation should be offset by a decreased vulnerability to invasive attacks.

1.1 Physical Random Functions

We now look more closely at what exactly a physical random function (PUF[1]) is. As its name suggests, it is a random function, i.e., a function whose outputs have been independently been drawn from some distribution. A PUF has the additional characteristic that it can only be evaluated with the help of a physical system. In a nutshell:

Definition 1. A *physical random function* is a random function that can only be evaluated with the help of a specific physical system. We call the inputs to a PUF *challenges*, and the outputs, *responses*.

To better understand what a PUF is, we consider a few example implementations:

Digital PUFs are conceptually the simplest kind of PUF. A digital secret key `K` is embedded in a tamper-proof package, along with some logic that computes `Response = RF(K, Challenge)`, where `RF` is some random function. Whenever a `Challenge` is given to the device, it outputs the corresponding `Response`.

Such a device is a PUF. Possessing the device allows one to easily get responses from challenges, but the tamper proof package prevents attackers from getting `K` and forging responses. However, it is not a compelling PUF as it relies on the secrecy of the digital secret `K`, which is precisely what we would like to avoid by using PUFs.

Optical PUFs were originally proposed by Ravikanth et al. [2002], Ravikanth [2001]. They are made up of a transparent optical medium containing bubbles. Shining a laser beam through the medium produces a speckle pattern (the response) behind the medium that depends on the exact position and direction of the incoming beam (the challenge). A study has also been made of the information content of an optical PUF [Tuyls et al. 2005].

Silicon PUFs were studied by Gassend [2003], Gassend et al. [2002b], Gassend et al. [2004], Lee et al. [2004], and Lim et al. [2005]. In this case,

[1]PUF avoids confusion with pseudorandom function and is the acronym that has been widely published. Some authors use the term physical unclonable function, but we find this term less accurate than physical random function.

the response is related to the time it takes for signals to propagate through a complex circuit. The challenge is an input to the circuit that reconfigures the path that signals follow through the circuit.

Both optical and silicon PUFs generate responses from physical systems which are difficult to characterize and analyze. They rely on the difficulty of taking a complex physical system, extracting all necessary parameters from it, and simulating it to predict responses. A sufficiently advanced attacker should be able to break the PUF, by following this path. However, we expect the difficulty to be considerably greater than the difficulty of extracting a digital secret from a device.

Another interesting point about these PUF implementations is that the PUF arises from random manufacturing variations, such as bubble position or exact wire delays. Consequently, the PUF is even a mystery for the manufacturer of the PUF. Also, manufacturers cannot make two identical PUFs even if they want to. We call these PUFs manufacturer resistant. In the case of silicon PUFs, manufacturers make every effort to reduce manufacturing variations in order to improve yield and performance. Despite these efforts, relative variations continue to increase as new process technologies are introduced (Chapter 14 of Chinnery and Keutzer [2002]). By checking that a manufacturer is using the high-density processes that are available, we can, therefore, be confident that manufacturer resistance holds and will continue to hold.

Moreover, these PUF implementations are relatively inexpensive to produce. In particular, silicon PUFs can be realized on standard CMOS technology [Weste and Eshraghian 1985], potentially making them more attractive than EPROM for identification of integrated circuits. Indeed, circuits with EPROM require extra processing steps which drive up the cost of a chip.

One difficulty with optical and silicon PUFs is that their output is noisy. Therefore, error correction, which does not compromise security, is required to make them noise-free. This problem has been considered elsewhere [Lim 2004; Suh et al. 2005], and will not be discussed in the rest of this paper.

The canonical application for PUFs is to use them as keycards [Ravikanth 2001]. In this application, a lock is initially introduced to a PUF and stores a database of challenge-response pairs (CRPs) corresponding to that PUF. Later, when the bearer of the PUF wants to open the lock, the lock selects one of the challenges it knows and asks the PUF for the corresponding response. If the response matches the stored response, the lock opens. In this protocol, CRPs can be used only once, so the lock eventually runs out of CRPs. This enables a denial-of-service attack, in which an adversary uses up all the lock's CRPs by repeatedly presenting it with an incorrect PUF. Because of this limitation, the keycard application is not very compelling. Nevertheless, it is all that can be done with a PUF, until we make it into a controlled physical random function.

1.2 Controlled Physical Random Functions

Definition 2. A *controlled physical random function* (CPUF) is a PUF that has been bound with an algorithm in such a way that it can only be accessed through a specific API.

As we shall see in Section 2.2, the main problem with uncontrolled PUFs is that anybody can query the PUF for the response to any challenge. To engage in cryptography with a PUF device, a user who knows a CRP has to use the fact that only he and the device know the response to the user's challenge. However, to exploit that fact, the user has to tell the device his challenge so that it can get the response. The challenge has to be told in the clear, because there is no key as yet. Thus, a man in the middle can hear the challenge, get the response from the PUF device, and use it to spoof the PUF device (this attack is detailed in Section 2.2).

Clearly the problem in this attack is that the adversary can freely query the PUF to get the response to the user's challenge. By using a CPUF in which access to the PUF is restricted by a control algorithm, this attack can be prevented. The API through which the PUF is accessed should prevent the man-in-the-middle attack we have described without imposing unnecessary limitations on applications.

A key contribution of this paper is the description in Section 2 of a simple, but very general, API for limiting access to a PUF. Some interesting properties of our API are:

- Anybody who knows a CRP that nobody else knows, can interact with the CPUF device to obtain an arbitrary number of other CRPs that nobody else knows. Thus users are not limited to using a small number of digital outputs from the PUF. Moreover, if one of these new CRPs was revealed to an adversary, transactions that use the other CRPs are not compromised. This is analogous to key management schemes that uses session keys derived from a master key.
- Anybody can use a CRP that only they know to establish a shared secret with the PUF device. Having a shared secret with the PUF device enables a wide variety of standard cryptographic primitives to be used.
- The control algorithm is deterministic. Since hardware random number generators are sensitive and prone to attack, being able to avoid them is advantageous.
- The only cryptographic primitive that needs to be built into the control algorithm is a collision-resistant hash function. All other cryptographic primitives can be updated during the lifetime of the CPUF device.

By selecting an appropriate API, a CPUF device can be resistant to protocol attacks. With careful design, optical and silicon PUFs can be made in such a way that the chip containing the control logic is physically embedded within the PUF: the chip can be embedded within the bubble-containing medium of an optical PUF, or the delay wires of a silicon PUF can form a cage on the top layer of the chip. This embedding should make probing of the control logic considerably more difficult, as an invasive attacker will have to access the wires to be probed without changing the response of the surrounding PUF medium. As we have illustrated in Figure 1, the PUF and its control logic have complementary roles. The PUF protects the control logic from invasive attacks, while the control logic protects the PUF from protocol attacks. This synergy

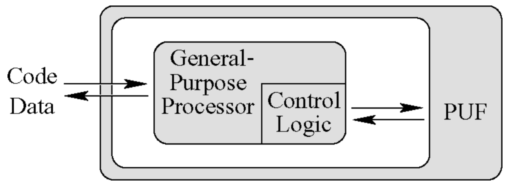


Fig. 1. We model a PUF device as a general-purpose processing element with possibly restricted access to a PUF. The processing element protects the PUF from protocol attacks, while the PUF protects the processing element from invasive attacks.

makes a CPUF far more secure than either the PUF or the control logic taken independently.

Based on the examples given above, we assume, in the remainder of this paper, that PUFs exist and we focus on designing an API, together with protocols, that allow a chain of trust to be established between the CPUF manufacturer and a party that wishes to interact securely with the PUF device. Our work extends Gassend et al. [2002a], where we introduced CPUFs. Here, we use hash blocks in our API design and we give more advanced application protocols.

1.3 Applications

There are many applications for which CPUFs can be used; we describe a few examples here. Other applications can be imagined by studying the literature on secure coprocessors, in particular Yee [1994]. We note that the general applications for which this technology can be used include all current applications in which there is a single symmetric key on a chip.

The easiest application is in smartcards, which implement authentication. Current smartcards have hidden digital keys that can be extracted using various attacks [Anderson 2001]. With a unique PUF on the smartcard that can be used to authenticate the chip, a digital key is not required: the smartcard *hardware* is itself the secret key. This key cannot be duplicated, so a person can lose control of a smartcard, retrieve it, and continue using it. Currently, the card should be canceled and a new one made because somebody might have cloned the card while it was out of its owner's control.

A bank could use certified execution to authenticate messages from PUF smartcards. This guarantees that the message the bank receives originated from the smartcard. It does not, however authenticate the bearer of the smartcard. Some other means, such as a PIN number or biometrics, must be used by the smartcard to determine if its bearer is allowed to use it. If the privacy of the smartcard's message is a requirement, then the message can also be encrypted.

A second application is for computers that implement private storage [Carroll et al. 2002; Alves and Felton 2004; Microsoft ; Trusted Computing Group 2004; Lie et al. 2000; Lie 2003; Suh et al. 2003]. A program wishing to store encrypted data in untrusted memory uses an encryption key, which depends uniquely on the PUF and its program hash. This requires a CPUF in order to accomplish the unique dependency. This idea is implemented in the AEGIS processor [Suh et al. 2003, 2005].

These computers can be used in grid computation, where the spare computation of thousands of machines on the Internet is pooled to solve computationally challenging problems [SETI@Home ; Distributed.Net]. One issue in grid computation is ensuring that the computation has been correctly carried out by the computers on the grid. This is especially important if people are paid for the computing power they put on the grid, as there is pressure to provide fake results in order to increase revenue.

In Section 4 we show how programs can generate common secret keys, which no user or other program can retrieve. This concept has applications in digital rights management and software licensing. It can also be used to create shared private memory.

1.4 Outline

Thus far, we have seen what a CPUF is, without detailing what API should be used to access the PUF. This API is described in Section 2, where we present a general-purpose CPUF device, how it is programmed, and how it can access the PUF. Then, in Section 3, we attack the problem of CRP management: how does a user who wants to use a CPUF device get a CRP that he trusts for that device? Finally, Section 4 shows some basic protocols that can be used by applications, which interact with a CPUF device.

2. CPUF PRIMITIVES

In the introduction section, we saw that a PUF can be used as a keycard. This application was proposed in Ravikanth [2001] and it is the only application for a PUF without control. In this section, we see why other applications fail and show how control can be used to make them work.

2.1 The Execution Environment

Without loss of generality, we model a PUF device as a general-purpose processing element that has access to a PUF (see Figure 1). The device will run any program that is given to it by the outside world. Initially, we consider that the processing element has unrestricted access to the PUF, but, in Section 2.2, we shall see that some restrictions need to be applied. This model does not imply that all PUF devices must actually be general-computing devices. In practice, one might want to implement only a limited set of functions on the device and hardwire them.

We make the assumption that programs running on the PUF device execute in a private and authentic way. That is, their internal data is inaccessible to an attacker and nobody can cause the program to execute incorrectly. This assumption is not trivial. We can partially justify it by the fact that the CPUF should be designed so that the logic functionality of the device is embedded within the physical system of the PUF and many kinds of invasive attacks will destroy the PUF before giving the adversary access to the logic. In Gassend [2003], we have, in addition, considered the “open-once” model in which the adversary can gain access to a single snapshot of internal variables, while breaking the PUF.

We now give a brief description of the execution environment that we use. The reader may want to refer back to this description later in the paper.

Programs will be described using a syntax close to C. A few notable changes are:

- We will often be identifying a piece of code by its hash. In the code, we indicate what region the hash should cover by using the `hashblock` keyword. During execution, whenever a hash block is reached, a hash is computed over its arguments, and stored in a special system register `PHashReg` that cannot be directly modified by the program. More precisely, each hash block has two sets of arguments: variable and code. `PHashReg` is a hash of all the arguments of the hash block concatenated together, with program block in the code section replaced by its hash. We assume that all the hashing is done using a cryptographically strong hash function. Moreover, concatenation is done in a way that ensures that a different sequence of arguments will always produce a different string to hash. Any code arguments to the hash block that are executable are then executed. When execution leaves the hash block, the previous value of `PHashReg` is restored (popped off an authentically stored stack). The need for hash blocks will become apparent in Section 2.3. The elaborate way in which the hashes are constructed will be fully used in Section 4.2.
- We declare variables using the keyword `my` as in the Perl language. We do this to avoid worrying about types, while still specifying where the variable is declared. Variables declared within a hash block are automatically cleared upon exit from the hash block. This way, data in variables that are declared in a hash block cannot be accessed once the block's execution has completed.

An example illustrates the use of `hashblock` and `my`. Note that we have given each hash block a name (A or B). This name will be used to designate the code and variables of a particular hash block when explaining how programs work. We have also prefixed lines within a hash block with the name of the hash block to clarify its extent.

```
 1      Foo(Bar)
 2      {
 3                        // PHashReg has value inherited from caller.
 4 A      hashblock(Bar)(  // Start of hashblock A
 5 A      {
 6 A                      // PHashReg is Hash(Bar; Hash(code in lines 6 to 14), Dummy).
 7 A        my FooBar = Bar / 2;
 8 A B      hashblock()( // Start of hashblock B
 9 A B      {
10 A B                     // PHashReg is Hash(; Hash(code in lines 10 to 11)).
11 A B        my BarFoo = FooBar + 3;
12 A B      });
13 A                       // PHashReg is Hash(Bar; Hash(code in lines 6 to 14), Dummy).
14 A                       // The value of BarFoo has been purged from memory.
15 A      }, Dummy);
16                         // PHashReg has value inherited from caller.
17                        // The value of FooBar has been purged from memory.
18     }
```

A number of cryptographic primitives will be used in what follows, including:

- `MAC(message, key)` produces a message authentication code (MAC) of `message` with a `key`.
- `EncryptAndMAC(message, key)` is used to encrypt and MAC `message` with a `key`.
- `PublicEncrypt(message, key)` is used to encrypt `message` with the public `key`.
- `Decrypt(message, key)` is used to decrypt a `message` that was encrypted with `key`.
- `PHash(HB)` is the value of `PHashReg` in the hash block denoted by `HB`. For example, `A` represents `(Bar; Hash(code from lines 6 to 14), Dummy)`. In this sequence, the comma (`,`) and the semicolon (`;`) represent different concatenations so that variable and code arguments can be distinguished. While the code from lines 6 to 14 is executing, `PHashReg` contains `PHash(A)`.

  Of these primitives, only `PHash` has to be permanently built into the CPUF device, as it participates in the generation of program hashes. All the other primitives can be updated. The only requirement is that the code of the primitives participates in the generation of program hashes. In an actual implementation `PHash` would, therefore, incorporate a hash of the primitives. Likewise, real programs will make use of libraries that will also need to be incorporated into the program hash. For the rest of this paper we ignore this issue and assume that all the primitives are hard coded. The problem of authenticating libraries is a common problem for code attestation and has been studied in detail by the trusted computing initiative [Trusted Computing Group 2004].

2.2 The Man-in-the-Middle Attack

Suppose that Alice wants to perform a computation on a device containing a PUF. She knows a set of CRPs for the PUF, and would like to know the result of the computation. Unfortunately, she is communicating with the device over an untrusted channel to which Oscar has access. Oscar would like Alice to accept an incorrect result as coming from the device. This example, as we shall see, captures the limitations of uncontrolled PUFs.

Alice attempts the following method:

1. She picks one of her CRPs `(Chal, Response)` at random.
2. She executes `GetAuthenticBroken(Chal)` on the PUF device (it is sent in the clear and without authentication).

```
GetAuthenticBroken(Chal)
{
  my Resp = PUF(Chal);
  ... Do some computation, produce Result ...
  return (Result, MAC(Result, Resp));
}
```

3. Alice uses the MAC and `Response` to check that the data she receives is authentic.

Unfortunately, this protocol does not work as Oscar can carry out a man-in-the-middle attack:

1. He intercepts the message in which Alice sends `GetAuthenticBroken` to the PUF and extracts `Chal` from it.
2. He executes `StealResponse(Chal)` on the PUF device.

```
StealResponse(Chal)
{
  return PUF(Chal);
}
```

3. Now that he knows the `Response`, he sends Alice the message `MAC(FakeResult, Response)`.
4. Since the MAC was computed with the correct response, Alice accepts `FakeResult` as valid.

The problem here is that as soon as Alice releases her challenge, Oscar can simply ask the PUF for the corresponding response and can then impersonate the PUF. As long as the PUF is willing to freely give out the response to challenges, this problem will persist.

2.3 The `GetSecret` Primitive

To solve this problem, we will move from a PUF that is wide open to one that is severely restricted. Suppose that we will only allow access to the PUF via a primitive called `GetSecret`, defined by:

```
GetSecret(Chal)=Hash(PHashReg, PUF(Chal))
```

This primitive is designed so that the CPUF device will not reveal a response to anybody. Instead it will reveal a combination of the response and the program that is being executed, that cannot be used to recover the response, because we assume that `Hash` is a one-way function.

Alice changes her `GetAuthenticBroken` program in the following way:

```
  GetAuthentic(Chal)
  {
HB  hashblock()(//Start of hashblock HB
HB  {
HB    my Result;
HB    hashblock ()(
HB    {
HB      ... Do some computation, produce Result...
HB    });
HB    my Secret = GetSecret(Chal);
HB    return (Result, MAC(Result, Secret));
HB  });
  }
```

Alice computes `Secret` from `Response` by computing `Hash(PHash(HB), Response)`, which allows her to check the MAC. Oscar, on the other hand, is now stuck. He has no way of getting `Secret`. If he sends a program of his own and calls `GetSecret`, he will get a different secret from the one Alice's program got. Also, `GetAuthentic` is well written; it does not leak `Secret` to the outside world (the inner hash block prevents the computation from producing/revealing `Secret`). Thus, Alice is now certain when the MAC check passes that she is looking at the result that is computed by the PUF device.

2.4 The `GetCRP` Primitive

Unfortunately for Alice, the latest version of the CPUF device is too restrictive. If Alice has a CRP then she can, indeed, interact securely with the device, but there is no way for her to get that CRP in the first place, since the device will never reveal a Response.

To solve this problem, we slightly lighten the restrictions that are placed on access to the PUF. For example, we could add a primitive called `GetCRP`. This primitive would pick a random challenge, compute its response, and return the newly found CRP to the caller. In this way, anybody would have access to an arbitrary number of CRPs, but as long as the space of challenges is large enough, the probability that the same CRP will be given to two different people is extremely small.

2.5 The `GetResponse` Primitive

In theory, a secure system could be built from `GetCRP`. In practice, however, random number generators are often vulnerable to attack. Since the scheme relies heavily on the strength of the random number generator, it would be nice to see if an alternative solution exists that is deterministic.

That solution does indeed exist. We replace `GetCRP` by `GetResponse`, which is defined by

$$\texttt{GetResponse()=PUF(PHashReg)}. \tag{1}$$

This way, anybody can generate a CRP `(PHashReg, GetResponse())`, but because of the hash function with which `PHashReg` is computed, nobody can choose to generate a specific CRP. The code of the hash block contributes to `PHashReg` so that the CRP that is generated is program dependent.

The challenge that is generated in `GetResponse` is equal to `PHashReg`. It depends on the code contained within the hash block, as well as on the variables that are arguments to the hash block. Often, one of these variables will simply be a nonce, that we call *prechallenge*, because it is used to determine the challenge.

Figure 2 summarizes the possible ways of going between prechallenges, challenges, responses, and secrets. In this diagram, moving down is easy. You just have to calculate a few hashes. Moving up is hard because it involves inverting one-way hashes. Going from left to right is easy for the program whose hash is used in the `GetResponse` or `GetSecret` primitives and hard for all other programs. Going from right to left is hard because the PUF is hard to invert.

The man-in-the-middle attack is prevented by each user having her own list of CRPs, where the challenges can be public, but the responses have to be private. From Figure 2, we see that to get his hands on a secret produced by some hash block `GSH`, an adversary has three options: he can be told the secret,

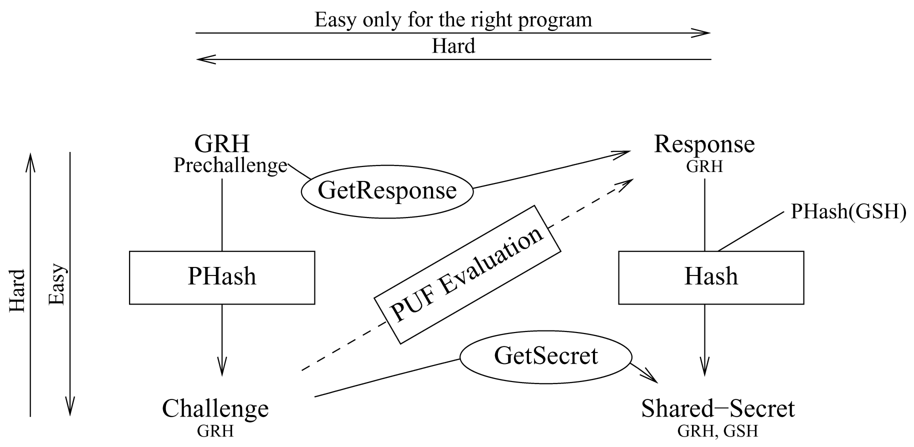


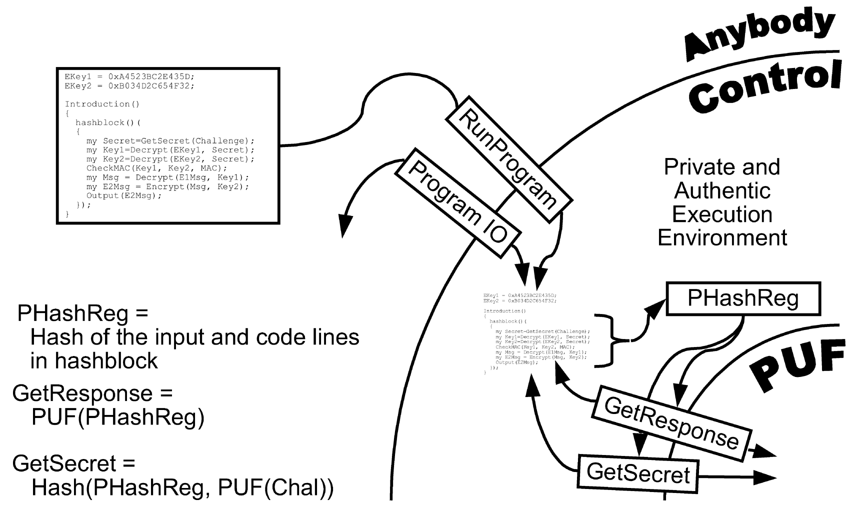
Fig. 2. This diagram shows the ways of moving between prechallenges, challenges, responses, and secrets. Rectangles show elementary operations that can be performed. Ellipses represent operations that are made by composing elementary operations. The dotted arrow indicates that because the PUF is controlled, nobody can actually evaluate it directly. GRH and GSH are the hash blocks that call `GetResponse` and `GetSecret`, respectively. If GRH or GSH appears below a value then that value depends on one of these hash blocks.

he can use `GetSecret` from within `GSH`, or he can hash an appropriate response. If the user does not tell the secret to the adversary and `GSH` does not leak the secret, then only the third option is possible. Thus, the adversary has to get his hands on the response. There are only two ways for him to do so: he can be told the response, or he can use `GetResponse` from within the hash block `GRH` in which it was created. If the user does not tell the adversary the response and if `GRH` does not leak the response then the adversary is out of options. In Section 3, we will see how `GRH` can be designed to use encryption to get the response to the legitimate user without leaking it to the adversary.

With the two primitives `GetSecret` and `GetResponse` that we have introduced, anybody can generate CRPs and use them to generate a secret value that is known only to them and to a program running on the PUF device. This choice of primitives is not unique, but we believe that this combination is particularly satisfying. Indeed, they make no assumptions on what use will be made of the secret value once it is generated. Moreover, in addition to the PUF itself, the only cryptographic primitive that needs to be built into the device is a hash function for `GetSecret` and `GetResponse` to be implemented. Figure 3 shows all the key elements of our generic CPUF device.

3. CHALLENGE–RESPONSE PAIR MANAGEMENT

In Section 2, we saw that access to the PUF had to be limited in order to prevent man-in-the-middle attacks. We proposed to make the PUF accessible only by the two primitives `GetResponse` and `GetSecret`. This places just the right amount of restriction on access to the PUF. The man-in-the-middle attack is thwarted, while anybody can gain access to a CRP and interact securely with the PUF.


Anybody
Control
Private and Authentic Execution Environment
RunProgram
Program IO
PHashReg
PUF
GetResponse
GetSecret
PHashReg = Hash of the input and code lines in hashblock
GetResponse = PUF(PHashReg)
GetSecret = Hash(PHashReg, PUF(Chal))

```
EKey1 = 0xA4523BC2E435D;
EKey2 = 0xB034D2C654F32;

Introduction()
{
  hashblock()(
  {
    my Secret=GetSecret(Challenge);
    my Key1=Decrypt(EKey1, Secret);
    my Key2=Decrypt(EKey2, Secret);
    CheckMAC(Key1, Key2, MAC);
    my Msg = Decrypt(E1Msg, Key1);
    my E2Msg = Encrypt(Msg, Key2);
    Output(E2Msg);
  });
}
```

Fig. 3. Summary of our CPUF device model. Anybody can give the device a program to run. Programs that run can access the device's PUF via the `GetSecret` and `GetResponse` primitives, which will take into account the `PHashReg` of the current program.

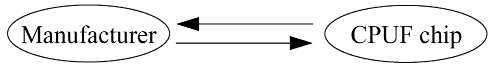


Fig. 4. Model for bootstrapping.

In this section, we go into the details of how a flexible trust infrastructure can be built in this model. It allows a chain of trust to be built from the manufacturer of a CPUF device to the end user, via an arbitrary sequence of certification authorities (that we shall call certifiers). First, just after the CPUF device is manufactured, the manufacturer uses `Bootstrap` in a secure factory environment to get a CRP for the device. The manufacturer can then use `Introduction` to provide CRPs to certification authorities, who, in turn, can provide them to end users. Anybody who has a CRP can use `Renew` to generate more CRPs. This is analogous to using keys derived from a master key. An attacker who manages to obtain a small number of CRPs (or derived keys) is unable to completely break the security of the device.

3.1 Bootstrapping

Bootstrapping is the most straightforward way to obtain a CRP. It is illustrated in Figure 4. No encryption or authentication is built into the protocol, as it is designed for users who are in physical possession of the device, and who can, therefore, directly hold a secure communication with the device. A CPUF device manufacturer would use bootstrapping to get CRPs for devices that have just been produced so that their origin can later be verified.

1. The user who wishes to get a CRP picks a prechallenge `PreChal` at random.
2. The user executes `Bootstrap(PreChal)` on the CPUF device.

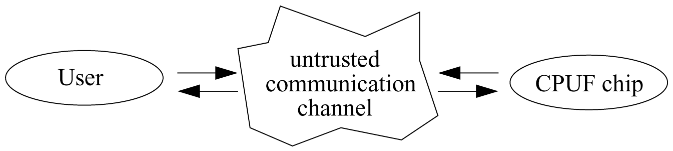


Fig. 5. Model for renewal.

```
     Bootstrap(PreChal)
     {
HB     hashblock (PreChal)( //Start of hashblock HB
HB     {
HB       return GetResponse();
HB     });
     }
```

3. The user gets the challenge of his newly created CRP by calculating `PHash(HB)`; the response is the output of the program.

If an adversary gets to know `PreChal`, he can replay the bootstrapping program with `PreChal` as input to obtain the corresponding CRP. Therefore, the user should discard `PreChal` after use. If `PreChal` is not known, then the security of the bootstrapping program relies on the one-wayness of the hash function with which `PHashReg=PHash(HB)` is computed.

3.2 Renewal

Renewal is a process by which a user who has a CRP for a CPUF can generate more CRPs over an untrusted network. This is illustrated in Figure 5. In this protocol, the user uses his CRP to create a secure channel from the CPUF device to himself and the CPUF sends the new response over that channel.

1. The user who wishes to generate a new CRP picks a prechallenge `PreChal` at random.
2. The user executes `Renew(OldChal, PreChal)` on the CPUF device, where `OldChal` is the challenge of the CRP which the user already knows.

```
    Renew(OldChal, PreChal)
    {
HB   hashblock (OldChal, PreChal)( //Start of hashblock HB
HB   {
HB     my NewResponse = GetResponse();
HB     my Secret = GetSecret(OldChal);
HB     return EncryptAndMAC(NewResponse, Secret);
HB   });
    }
```

3. The user computes `Hash(PHash(HB), OldResponse)` to calculate `Secret`, and uses it to check the MAC and retrieve `NewResponse`. The new challenge is computed by `PHash(HB)`.

Note that `PreChal` is included in the hash block so that it participates in the generation of `Secret` and, therefore, in the MAC. Thus, if any tampering occurs with `PreChal` it will be detected in Step 3.

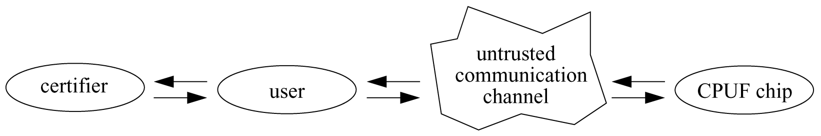


Fig. 6. Model for introduction.

OldChal is also, included in the hash block so that it participates in the generation of NewResponse. Thus, an adversary cannot retrieve NewResponse by replaying the renew program with inputs PreChal and an old challenge different from OldChal for which he knows the corresponding response.

The security of renewal relies on whether the response corresponding to OldChal is only known to the user. Note that the use of PreChal allows the user to derive multiple new CRPs by using renewal with the same OldChal. The corresponding responses are only known to the user if the response to OldChal is not compromised.

In Torlak et al. [2006], we encoded the renewal protocol in the Alloy modeling language [Jackson 2002] and checked its security using the Alloy Analyzer [Jackson 2000]. The protocol was found to be secure against replay and parallel session attacks for an unbounded number of sessions.

3.3 Introduction

Introduction is an operation that allows a user who has a CRP for a CPUF to provide a CRP to another user, assuming that there is a trusted channel between the users (established using public key cryptography, for example), but that communication with the PUF device is untrusted. It is illustrated in Figure 6. This operation could be used, for example, by a user who wants to be sure that he is getting a CRP to a genuine CPUF device.

In a real-world deployment, a chain of trust would be built from the manufacturer to the end user of the CPUF device. The manufacturer would collect CRPs on the production line using bootstrapping and use them to introduce the PUF to various competing certifiers. The end user would then ask a certifier of his choosing to introduce him to the device.

Many variants of introduction are possible, depending on how much trust the user is willing to place in the certifier. Here we present our strongest version, which uses public key cryptography and protects against passive attacks (i.e., attacks in which the certifier gives the user a valid CRP, but keeps a copy of it to eavesdrop on the user). We do not, however, protect against active attacks in which the certifier simply gives the user an incorrect CRP and makes him think he is talking to a PUF when, in fact, he is not. This is the minimum amount of trust that the user must place in the certifier in order to make introduction possible.

With this protocol, the user can also get introduced to the same CPUF device by many different certifiers. They would all have to actively attack him in a consistent way for the attack to succeed. In this protocol, the user needs to have a public key PubKey that gets used to encrypt the new response. The user would run `Introduction` multiple times with the same `PubKey` and `PreChal`, but a different `OldChal` for each certifier.

1. The certification authority picks `(OldChal, OldResponse)` from one of its CRPs and computes `Secret` as `Hash(PHash(HB), OldResponse)`, where `HB` is the hash block in the introduction program given in Step 2. The certification authority gives `(OldChal,Secret)` to the user.
2. The user picks a prechallenge `PreChal` at random. The user, next executes `Introduction(OldChal, PubKey, PreChal)`.

```
     Introduction(OldChal, PubKey, PreChal)
     {
  HB  hash block (PubKey, PreChal)(//Start of hashblock HB
  HB  {
  HB    my NewResponse = GetResponse();
  HB    my Message = PublicEncrypt(NewResponse, PubKey);
  HB    my Secret = GetSecret(OldChal);
  HB    return (Message, MAC(Message, Secret));
  HB  });
     }
```

3. The user uses `Secret` to check the MAC. Notice that the user and the `Introduction` program agree on the value of `Secret`, because, in each case, it is computed from `Hash(PHash(HB), OldResponse)`. The user decrypts `Message` with his private key to get the response and computes `PHash(HB)` to get the challenge.

In the protocol, the certification authority does not communicate with the PUF. We prefer for the user to communicate with the PUF, because he will often be the one who has a channel open to the PUF.

Since `NewResponse` is encrypted with `PubKey`, an adversary cannot directly use it to get `NewResponse`. Moreover, `PubKey` is included in the hash block so that it participates in the generation of `NewResponse`. Thus, an adversary cannot retrieve `NewResponse` by replaying the introduction program with his own public key, as changing the public key changes the secret that is produced. Even the certifier, who knows `(OldChal, OldResponse)`, cannot retrieve `NewResponse`.

There are other ways of doing introduction. For example, a slightly different protocol was used in Gassend [2003] and Gassend et al. [2002a]. In that protocol, the certifier gives `OldResponse` to the user, who can then compute `Secret` himself. The user then has to perform a *private renewal* that uses public key cryptography, to prevent passive attacks from the certification authority. In this case, the certifier cannot use the CRP `(OldChal, OldResponse)` a second time. For devices with very limited computation resources, it may be necessary to forgo public key cryptography altogether. In this case, the certifier introduces the user to the device by giving him a CRP. However, the user cannot then prevent passive attacks from the certifier.

4. APPLICATION PROTOCOLS

In certified execution, a certificate is produced that guarantees to the user that the program was run without being tampered with on a processor. We first discuss how a certificate is produced that can only be verified by the user of the processor. It relies on a shared secret between the user and the processor with the PUF. Next, we discuss how a certificate can be produced that can be verified by any third party. We call such a certificate a proof of execution. It relies on the ability to share a secret between different programs.

4.1 Certified Execution

It is possible for a certificate to be produced that proves to the user of a specific CPUF, that a specific computation was carried out on this CPUF and that the computation produced a given result. The person requesting the computation can then rely on the trustworthiness of the CPUF manufacturer who can vouch that he produced the CPUF, instead of relying on the owner of the CPUF. We call this *certified execution*. It is essentially the same as `GetAuthentic`, which was presented in Section 2.3 (in fact, `Renewal` and `Introduction` are also special cases of certified execution).

1. The user who wishes to run a program with program code `Prog` on a specific CPUF picks one of the CRPs `(Challenge, Response)` he knows for that CPUF.
2. The user executes `CertifiedExecution(Challenge, Prog)` on the CPUF device.

```
         CertifiedExecution(Challenge, Prog)
         {
 HB         hashblock (Prog)(// Start of hashblock HB
 HB         {
 HB           my Result;
 HB HA        hashblock ()(   // Start of hashblock HA
 HB HA        {
 HB HA          Result = RunProg(Prog);
 HB HA        });
 HB           my Secret = GetSecret(Challenge);
 HB           my Certificate = (Result, MAC(Result, Secret));
 HB           Return Certificate;
 HB         });
         }
```

3. The user computes `Hash(PHash(HB), Response)` to calculate `Secret` and uses it to check the MAC in `Certificate` and to accept the returned `Result` as authentic.

Notice that `Secret` depends on `Prog`. This means that tampering with `Prog` will be detected so that if the returned `Result` is authentic then it is equal to the output of `Prog`.

In this application, the user is trusting that the CPUF performs the computation correctly. This is easier to ensure if all the resources used to perform the computation (memory, CPU, etc.) are on the same CPUF device, and included in the PUF characterization. In Suh et al. [2003, 2005] a more sophisticated architecture is discussed in which a chip can securely utilize off-chip resources.

It uses ideas from Lie et al. [2000] and it utilizes a memory integrity scheme that can be implemented in a hardware processor [Gassend et al. 2003].

It is also possible for a CPUF to use the capabilities of other networked CPUF devices. In that case, the CPUF has CRPs for each of the other CPUF devices it is using and performs computations using protocols similar to the one described in this section.

4.2 Joint Secret Key Generation

Thus far we have only considered sharing a secret between a user and a specific program. In some cases, it can be desirable to have a secret shared between two programs. The hash block mechanism makes this possible, as illustrated in the following example:

```
Prog1()
{
  hashblock()(
  {
    my Secret = GetResponse();
    ... Do something with Secret here ...
  }, Prog2Hash);
}

Prog2()
{
  hashblock()(Prog1Hash,
  {
    my Secret = GetResponse();
    ... Do something with Secret here ...
  });
}
```

In these programs, `Prog1Hash` and `Prog2Hash` are the hashes of the code in the hash blocks of `Prog1` and `Prog2`, respectively. Thus, in both programs `PHashReg= Hash(Prog1Hash, Prog2Hash)`, which leads to the same `Secret`. No other hash block could generate this value of `PHashReg`, so no other program (or user) can use `GetResponse` to obtain `Secret`.

This example can naturally be extended to sharing between more than two programs. If we wanted `Secret` to also be shared with a user who knows the CRP `(Challenge, Response)`, we could replace `GetResponse()` with `GetSecret(Challenge)`.

4.3 Proof of Execution

In certified execution the user can create certificates himself, since the user can compute `Secret`. However, the user cannot use `Certificate` as a proof of execution to third parties. It only serves as a certificate to himself. Of course, a third party may simulate the interaction of the user with the CPUF device. If `Prog` then, represents a deterministic algorithm, the third party may check the results it receives with the results the user claimed to have received. This approach is undesirable, because it needs to rerun `Prog` and it needs to simulate all the interactions between the user and `Prog`.

To create a proof of execution (e-proof), which is efficiently verifiable by any third party, we use joint secret key generation, as introduced in Section 4.2. An *execution program* computes an *e-proof* containing the program output and a MAC signed with the joint key. Anybody with a CRP can check the e-proof by running an *arbitration program* on the CPUF.

1. The user who wishes to run a program with program code `Prog` on a specific CPUF picks one of the CRPs `(Challenge, Response)` he knows for that CPUF.
2. The user computes `HCodeA=Hash(CodeA)`, where `CodeA` is the code representing the program code in hash block `HA` of `ArbitrationProgram` (see below). Next, the user executes `ExecutionProgram(Prog, HCodeA)` on the CPUF device.

```
     ExecutionProgram(Prog, HCodeA)
     {
     my HProg = Hash(Prog);
HE   hashblock (HProg)(HCodeA, // Start of hashblock HE
HE   {
HE                              // PHashReg is Hash(Hash(Prog); Hash(CodeA),
                                Hash(CodeE))
HE       my Result;
HE HB    hashblock (           //  Start of hashblock HB
HE HB    {
HE HB       Result = RunProg(Prog);
HE HB    });
HE       my Secret = GetResponse();
HE       my EProof = (Result, MAC(Result, Secret));
HE       return EProof;
HE   });
     }
```

To convince a third party (or himself) that the `EProof` has been generated by running `Prog` on the specific CPUF device, the user performs the following protocol.

1. The user transmits `Prog` and `EProof` to the third party.
2. The third party computes `HProg=Hash(Prog)` and `HCodeE=Hash(CodeE)` where `CodeE` is the program code in hash block `HE` of `ExecutionProgram` (the code for `ArbitrationProgram` and `ExecutionProgram` is public knowledge). He next executes `ArbitrationProgram(EProof, HCodeE, HProg)` on the CPUF device by using certified execution, as introduced in Section 4.1. That is, he runs `CertifiedExecution(Challenge, ArbitrationProgram(...))`.

```
   ArbitrationProgram(EProof, HCodeE, HProg)
   {
HA   hashblock (HProg)( // Start of hashblock HA
HA   {
HA                     // PHashReg is Hash(Hash(Prog); Hash(CodeA), Hash(CodeE))
HA     my (Result, M) = EProof;
HA     my Secret = GetResponse();
HA     If M = MAC(Result, Secret) return(true); else return(false);
HA   }, HCodeE);
  }
```

3. The third party uses the certificate generated by the certified execution protocol to check whether the received value of the output, `true` or `false`, is authentic. Only if the output is `true`, the third party is convinced that `EProof` was generated by `ExecutionProgram(Prog, HCodeA)` on the CPUF device. In other words, the third party is convinced that `Prog` was executed in `ExecutionProgram(Prog, HCodeA)` on the CPUF device.

Notice that both `ExecutionProgram` and the `ArbitrationProgram` compute the same `Secret` (we are doing pure computation that cannot become stale; any day we run the same computation it will give the same result). For this reason, the arbitration program will correctly verify the e-proof.

Since `Prog` is run within a separate hash block, `Prog` cannot leak `Secret`. Therefore, the only two programs that can compute `Secret` are the arbitration and execution programs. The user can only retrieve `Secret` by using `GetResponse()` within a program that hashes to `HCodeE` or `HCodeA`. Such a program is computationally intractable to find, because of the collision resistance of `Hash`.

In this protocol, the program output, `Result`, is given to the third party along with `EProof`, allowing the third party to verify that `Prog` has been run, and that it output `Result`. For some applications, it is desirable to keep all or part of `Result` hidden. In this case, only the (possibly empty) public part of `Result` should be included in the `EProof` computation.

5. CONCLUSION

In this paper, we have presented CPUFs, which are physical random functions that can only be accessed through a restricted API. CPUFs hold the promise of being a low cost way to increase the resistance to invasive attack of devices that participate in cryptographic protocols.

We have presented a particular API in which the CPUF device has a general-purpose processing element that can access a PUF using two primitives called `GetResponse` and `GetSecret`. These primitives use a hash of the program fragment that is being run to identify that program fragment. `GetResponse` allows anybody to get a large number of challenge-response pairs (CRPs), without letting anybody choose the specific CRP that somebody else is already using. `GetSecret` allows the bearer of a CRP to generate a secret value that is known only to himself and his program running on the CPUF device.

Using this API, we have shown how the identity of a CPUF device can be verified by its end user, using a few simple protocols. First the manufacturer uses `Bootstrap` to get a CRP from the PUF as it is sitting in the secure environment where it was made. The `Renew` and `Introduce` protocols can then be used to generate more CRPs and to give a CRP to another person, respectively. `Renew` and `Introduce` leverage the existing CRP to establish a secure channel between the CPUF device and the person interacting with it, relieving the need for the device to be in a secure environment after the initial `Bootstrap`. Using these protocols, the identity of the device can be passed from the device manufacturer to the end users via an arbitrary number of certification authorities.

Finally, we have presented a few building blocks that can be useful in applications. Certified execution allows a user to run a program on a CPUF device and be sure that the output he receives was indeed generated on a specific device. Proof of execution goes one step farther, allowing the user to prove to a third party that a program has been executed on a specific device. In proof of execution, we show how two programs running on the same CPUF device at different times can communicate securely together.

Throughout this paper, we have informally argued that the presented protocols are secure. A formal proof of certified execution, which is at the core of all the other protocols we have presented, can be found in Torlak et al. [2006]. We encoded the protocol in the Alloy-modeling language [Jackson 2002] and checked its security using the Alloy Analyzer [Jackson 2000]. The protocol was found to be secure against replay and parallel session attacks for an unbounded number of sessions.

Overall, we feel that CPUFs should allow the potential of physical random functions to be fully exploited. This should provide cheap devices with much increased resistance to invasive physical attacks, in a world where cryptography has become pervasive in our cell phones, PDAs, and even credit cards.

REFERENCES

ALVES, T. AND FELTON, D. 2004. Trustzone: Integrated hardware and software security. ARM. White paper.

ANDERSON, R. AND KUHN, M. 1996. Tamper resistance—A cautionary note. In *Proceedings of the Second Usenix Workshop on Electronic Commerce*. Usenix Association, Berkeley, CA. 1–11.

ANDERSON, R. AND KUHN, M. 1997. Low cost attacks on tamper resistant devices. In *IWSP: International Workshop on Security Protocols, LNCS*. Springer-Verlag, New York.

ANDERSON, R. J. 2001. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, New York.

CARROLL, A., JUAREZ, M., POLK, J., AND LEININGER, T. 2002. Microsoft “palladium”: A business overview. In *Microsoft Content Security Business Unit*.

CHINNERY, D. AND KEUTZER, K. 2002. *Closing the Gap Between ASIC & Custom*. Kluwer Academic Publi., Boston, MA.

DISTRIBUTED.NET. `http://distributed.net/`.

GASSEND, B. 2003. Physical Random Functions. M.S. thesis, Massachusetts Institute of Technology.

GASSEND, B., CLARKE, D., VAN DIJK, M., AND DEVADAS, S. 2002a. Controlled physical random functions . In *Proceedings of 18th Annual Computer Security Applications Conference*. Applied Computer Security Associates (ACSA), Silver Spring, MD.

GASSEND, B., CLARKE, D., VAN DIJK, M., AND DEVADAS, S. 2002b. Silicon physical random functions. In *Proceedings of the Computer and Communication Security Conference*. ACM, New York.

GASSEND, B., SUH, G. E., CLARKE, D., VAN DIJK, M., AND DEVADAS, S. 2003. Caches and Merkle trees for efficient memory integrity verification. In *Proceedings of Ninth International Symposium on High Performance Computer Architecture*. IEEE, New York.

GASSEND, B., LIM, D., CLARKE, D., VAN DIJK, M., AND DEVADAS, S. 2004. Identification and authentication of integrated circuits. *Concurrency and Computation: Practice and Experience 16*, 11, 1077–1098.

GUTMAN, P. 1996. Secure deletion of data from magnetic and solid-state memory. In *Sixth USENIX Security Symposium Proceedings*. Usenix Association, Berkeley, CA, 77–89.

JACKSON, D. 2000. Automating first-order relational logic. In *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering/European Software Engineering Conference (FSE/ESEC ’00)*. ACM, New York.

JACKSON, D. 2002. Alloy: A lightweight object modelling notation. *ACM TOSEM 11*, 2, 256–290.

KOCHER, P., JAFFE, J., AND JUN, B. 1999. Differential power analysis. *Lecture Notes in Computer Science 1666*, 388–397.

LEE, J.-W., LIM, D., GASSEND, B., SUH, G. E., VAN DIJK, M., AND DEVADAS, S. 2004. A technique to build a secret key in integrated circuits with identification and authentication applications. In *Proceedings of the IEEE VLSI Circuits Symposium*. IEEE, New York.

LIE, D. 2003. Architectural support for copy and tamper-resistant software. Ph.D. thesis, Stanford University.

LIE, D., THEKKATH, C., MITCHELL, M., LINCOLN, P., BONEH, D., MITCHELL, J., AND HOROWITZ, M. 2000. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*. 168–177.

LIM, D. 2004. Extracting Secret Keys from Integrated Circuits. M.S. thesis, Massachusetts Institute of Technology.

LIM, D., LEE, J. W., GASSEND, B., SUH, G. E., VAN DIJK, M., AND DEVADAS, S. 2005. Extracting secret keys from integrated circuits. *IEEE Trans. VLSI Syst. 13*, 10, 1200–1205.

MICROSOFT. Next-Generation Secure Computing Base. `http://www.microsoft.com/resources/ngscb/defaul.mspx`.

RAVIKANTH, P. S. 2001. Physical one-way functions. Ph.D. thesis, Massachusetts Institute of Technology.

RAVIKANTH, P. S., RECHT, B., TAYLOR, J., AND GERSHENFELD, N. 2002. Physical One-Way Functions. *Science 297*, 2026–2030.

SETI@HOME.

SKORIC, B., TUYLS, P., AND OPHEY, W. 2005. Robust key extraction from physical unclonable functions. In *Proceedings of the Applied Cryptography and Network Security Conference 2005*, J. Ionnidis, A. Keromytis, and M. Yung, Eds. Lecture Notes in Computer Science, vol. 3531. Springer-Verlag. New York. 407–422.

SMITH, S. W. AND WEINGART, S. H. 1999. Building a high-performance, programmable secure coprocessor. *Computer Networks (Special Issue on Computer Network Security) 31*, 8 (Apr.), 831–860.

SUH, G. E., CLARKE, D., GASSEND, B., VAN DIJK, M., AND DEVADAS, S. 2003. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th Int'l Conference on Supercomputing (MIT-CSAIL-CSG-Memo-474 is an updated version)*. ACM, New York.

SUH, G. E., O'DONNELL, C. W., SACHDEV, I., AND DEVADAS, S. 2005. Design and implementation of the AEGIS single-chip secure processor using physical random functions. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (MIT-CSAIL-CSG-Memo-483 is an updated version available at http://csg.csail.mit.edu/pubs/memos/Memo-483/Memo-483.pdf)*. ACM, New York.

TORLAK, E., VAN DIJK, M., GASSEND, B., JACKSON, D., AND DEVADAS, S. 2006. Knowledge flow analysis for security protocols. http://arxiv.org/abs/cs/0605109.

TRUSTED COMPUTING GROUP. 2004. TCG Specification Architecture Overview Revision 1.2. http://www.trustedcomputinggroup.com/home.

TUYLS, P., SKORIC, B., STALLINGA, S., AKKERMANS, A., AND OPHEY, W. 2005. Information theoretical security analysis of physical unclonable functions. In *Proceedings Conf on Financial Cryptography and Data Security 2005*, A. Patrick and M. Yung, Eds. Lecture Notes in Computer Science, vol. 3570. Springer-Verlag, New York. 141–155.

WESTE, N. AND ESHRAGHIAN, K. 1985. *Principles of CMOS VLSI Design: A Systems Perspective*. Addison-Wesley, Reading, PA.

YEE, B. S. 1994. Using secure coprocessors. Ph.D. thesis, Carnegie Mellon University.

Received October 2005; revised March 2007; accepted April 2007