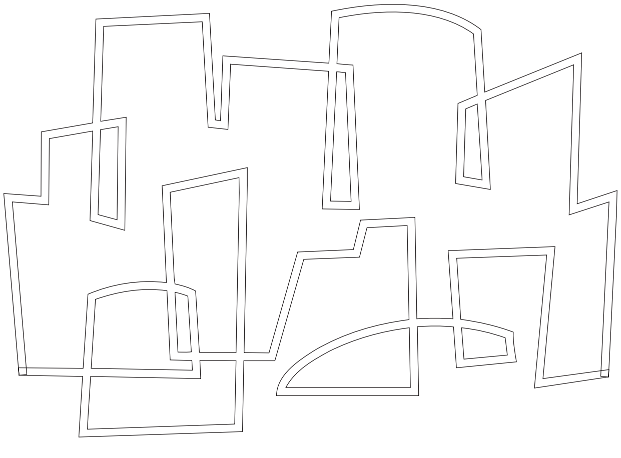
Physical Random Functions

Blaise Gassend

Masters Thesis

2003, February

Computation Structures Group
Memo 458



Physical Random Functions

by

Blaise L. P. Gassend

Diplôme d'Ingénieur
École Polytechnique, France, 2001

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2003

Author . . . .
Department of Electrical Engineering and Computer Science
January 17, 2003

Certified by . . . .
Srinivas Devadas
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Professor of Electrical Engineering and Computer Science
Chairman, Department Committee on Graduate Students

Physical Random Functions

by

Blaise L. P. Gassend

Submitted to the Department of Electrical Engineering and Computer Science
on January 17, 2003, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

In general, secure protocols assume that participants are able to maintain secret key information. In practice, this assumption is often incorrect as an increasing number of devices are vulnerable to physical attacks. Typical examples of vulnerable devices are smartcards and Automated Teller Machines.

To address this issue, Physical Random Functions are introduced. These are Random Functions that are physically tied to a particular device. To show that Physical Random Functions solve the initial problem, it must be shown that they can be made, and that it is possible to use them to provide secret keys for higher level protocols. Experiments with Field Programmable Gate Arrays are used to evaluate the feasibility of Physical Random Functions in silicon.

Thesis Supervisor: Srinivas Devadas
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

This work was funded by Acer Inc., Delta Electronics Inc., HP Corp., NTT Inc., Nokia Research Center, and Philips Research under the MIT Project Oxygen partnership.

There are also many people who have contributed to this project in big or little ways. I will mention a few of them, and certainly forget even more.

First of all, I would like to thank my advisor, Srinivas Devadas, for his boundless energy and incessant stream of ideas, which were essential in getting Physical Random Functions off the ground. I also warmly thank the other key players in the development of Physical Random Functions: Marten van Dijk, who is always full of enthusiasm when I come to talk to him; and Dwaine Clarke, who always makes me “spell it out” for him – often quite a fruitful exercise.

I also greatly appreciated the help of Tara Sainath and Ajay Sudan, who decided to spend their summer hacking away at FPGA test-boards, instead of going somewhere sunny as they rightly should have. My office-mates, Prabhat Jain, Tom Kotwal and Daihyun Lim, deserve special credit for putting up with me, despite my tendency to chat with them while they are trying to work, and write on my white-board with toxic smelling markers. My girl-friend, Valérie Leblanc, and my apartment-mate, Nate Carstens, have been very good at not pointing out that I haven’t been doing my share of kitchen work lately, as I devoted myself to writing. My gratitude also extends to all the people on my end of the second floor, who contribute to making it a warm and lively place. And finally, I thank my parents, and all my past professors, without whom I would never have ended up at MIT in the first place.

Last, but not least, I must state that this work would never have been possible but for Ben & Jerry’s ice-cream. Without it, surviving the lengthly meetings in which all these ideas first surfaced would have been impossible.

Contents

List of Figures

Chapter 1

Introduction

1.1 Storing Secrets

One of the central assumptions in cryptographic protocols is that participants are able to store secret keys. Protocol participants are able to protect themselves from adversaries because they know something that the adversary does not know. Encrypted messages work because their intended recipient knows a decryption key that eavesdroppers do not know. Digital signatures work because the signer knows some information that nobody else knows, so a potential impostor is unable to forge a signature.

In these examples, we can see that knowing secret information allows someone to perform a certain action (read a message, produce a signature, etc.). In a way the secret keys that are involved identify their bearer as being authorized to perform a certain action.

In many cases, unfortunately, keeping a secret is extremely hard to do. Many devices are placed in environments in which they are vulnerable to physical attack. Automated Teller Machines can be dismantled to try to extract keys that are used for PIN calculation or to communicate with the central bank. In the cable television industry, users can open their set-top boxes to try to extract decoding keys, so that others can get free access to premium rate channels. The EPROM on smartcards can be examined to extract the bits of their signing keys. Many examples and details of exploits can be found in [2, 4].

The state of the art method for protecting against key extraction through invasive physical attack is to enclose the key information in a tamper sensing device. A typical example of this is the IBM 4758 [23]. A battery operated circuit constantly monitors the device, using a mesh of wires that completely envelops it. If any of the wires are cut, the circuit immediately clears the memory that contains critical secrets. This type of protection is relatively expensive as the circuit must be enclosed in tamper sensing mesh and the tamper sensing circuitry must be continuously powered. Despite this, the circuit remains vulnerable to sophisticated attacks. For example, shaped charges could be used to separate the tamper sensing circuit from the memory it is supposed to erase, faster than the memory can be cleared.

The goal of this thesis is to explore a different way of managing secrets in tamper prone physical devices. It is based on the concept of Controlled Physical Random Functions (CPUF).[1] A Physical Random Function (PUF) is essentially a random function, which is

[1] Footnote 1 on page 17 explains the CPUF acronym.

bound to a physical device in such a way that it is computationally and physically infeasible to predict the output of the function without actually evaluating it on the original device. That is, taking the device apart, or trying to find patterns in the function's output won't help you predict the function's output on new inputs. A CPUF is a PUF that can only be evaluated only from within a specific algorithm.

Our hope is that with PUFs, a greater level of physical security will be attainable, at a lower cost than with the classical approach of storing a digital secret. Indeed, an attacker can easily read a digital secret by opening a device. To protect the secret, attackers must be prevented from opening the device, or the device must be able to detect invasive attacks and forget its secrets if an attack is detected. Our approach is different; we extract the secret from a complex physical system in such a way that the secret is hard to get by any other means. We can do this because we accept not to choose the secret, and we accept to do more work to reliably reconstruct the secret. In exchange, we can do away with most of the expensive protection mechanisms that were needed to protect the digital secret.

1.2 Related Work

The idea of PUFs is not new. It was first studied in [21] under the name of Physical One-Way Functions. In that work, a wafer of bubble filled transparent epoxy is used. When a laser beam is shone through the wafer and projected onto a light sensor, a speckle pattern is created. This pattern is a complicated function of the direction from which the laser beam is incident, and the configuration of the bubbles in the epoxy. For suitable bubble sizes and densities, it is hypothesized that such a wafer is unclonable and that the speckle pattern for a given illumination is hard to predict from the patterns for other illuminations.

What makes this approach work is that a sufficiently complex physical system can be hard to clone, and can be made to exhibit a hard to predict but repeatable behavior. Thus, we have a way of extracting some secret information that we do not choose from a physical system. It appears that not being able to choose the secret information that is extracted makes physical protection much cheaper to implement. This key observation explains why PUFs are easier to implement than secure digital key storage.

The major advantage of CPUFs over the work described in [21] resides in the control. Without control, the only possible application is the one-time pad identification system that is presented in Section 2.2. The output of the PUF cannot be used as a secret for higher level protocols because of the possibility of a man-in-the-middle attack: an adversary can monitor outputs from the PUF to get the device's secret for a specific instance of the protocol. He can then use that secret to run the higher level protocol himself, pretending to be the device.

In our implementation of CPUFs, we use a silicon Integrated Circuit (IC) as the complex physical system from which we seek to extract PUF data. During IC fabrication, a number of process variations contribute to making each integrated circuit unique [6, 8]. These process variations have previously been used to identify ICs. For example, [17] uses random fluctuations in drain currents to assign an identifier to ICs. However, the identification system that results is not resistant to adversarial presence. An adversary can easily find out the identification string for an IC and then masquerade as that IC, since he has all the available identification information.

Unlike the system we propose, these IC identification circuits attempt to extract information from the manufacturing variations in an extremely controlled way, for reliability reasons. In contrast, we use a complex physical circuit for which it is hard to predict the output from direct physical measurements. Moreover, we produce a function instead of a single value, so revealing one output of the function does not give away all the IC's identification information. The parameter that we measure in our circuit is the delay of a complex path through a circuit.

1.3 Organization

This thesis is structured as follows. Chapter 2 gives a general overview of PUFs. It includes definitions, a simple application, threat models and general remarks on PUF implementation.

Chapters 3 and 4 go further into the details of PUF implementation. First a weak PUF has to be made that is directly based on a complex physical system. Chapter 3 shows how optical and physical systems can be used to make a weak CPUF. Experimental results for silicon PUFs implemented on Field Programmable Gate Arrays (FPGA) show that the systems we suggest can actually be built, and give an idea of the orders of magnitude that are involved. The following chapter takes the weak CPUF and shows how to make it into a CPUF that is more secure and reliable. This improvement is made by surrounding the weak CPUF with some digital pre- and post-processing.

Now that we know how to build CPUFs, we look at how they can be used. Our main interest is to use PUFs in a network context; machines on the network could use their PUF to prove their physical identity and integrity, even if they are located in hostile environments. Ideally, we would like to build a PUF infrastructure to exchange credentials that has a comparable flexibility to public key infrastructures. Chapter 5 shows how such an infrastructure can be built. Attaching an unique identity to a device raises serious privacy concerns, which our infrastructure is able to address. The chapter is concluded with an example of use for distributed computation.

The network context is not the only context in which PUF ideas can be put to use. In Chapter 6 we show how the physical systems that we have used to make PUFs can instead be used to store keys on a device, in a way that is more secure than simply storing keys digitally.

This research has opened up many opportunities for future work. They are presented in Chapter 7 along with some concluding remarks.

Many terms are defined in this thesis and later assumed to be known to the reader. A glossary is provided at the end of the document to help the reader who has missed one of the definitions.

Chapter 2

Physical Random Functions

2.1 Definitions

First we give a few definitions related to Physical Random Functions. Because it is difficult to quantify the physical abilities of an adversary, these definitions will remain somewhat informal.

2.1.1 Physical Random Functions

Definition 1 *A Physical Random Function (PUF)*[1] *is a function that maps challenges to responses, that is embodied by a physical device, and that has the following properties:*

1. *Easy to evaluate: The physical device is capable of evaluating the function in a short amount of time.*

2. *Hard to characterize: From a limited number of plausible physical measurements or queries of chosen Challenge-Response Pairs (CRP), an attacker who no longer has the device, and who can only use a limited amount of resources (time, money, raw material, etc...) can only extract a negligible amount of information about the response to a randomly chosen challenge.*

In the above definition, the terms short and limited are relative to the size of the device, which is the security parameter. In particular, short can be read as linear or low degree polynomial, and limited can be read as polynomial. The term plausible is relative to the current state of the art in measurement techniques and is likely to change as improved methods are devised.

In previous literature [21], PUFs were referred to as Physical One Way Functions. We believe this terminology to be confusing because PUFs do not match the standard meaning of one way functions [19].

In the rest of this thesis we will often compare PUF methods with methods that involve storing and protecting a digital secret. We will generally refer to the latter methods as classical methods.

[1] PUF actually stands for Physical Unclonable Function. It has the advantage of being easier to pronounce, and avoids confusion with Pseudo-Random Functions.

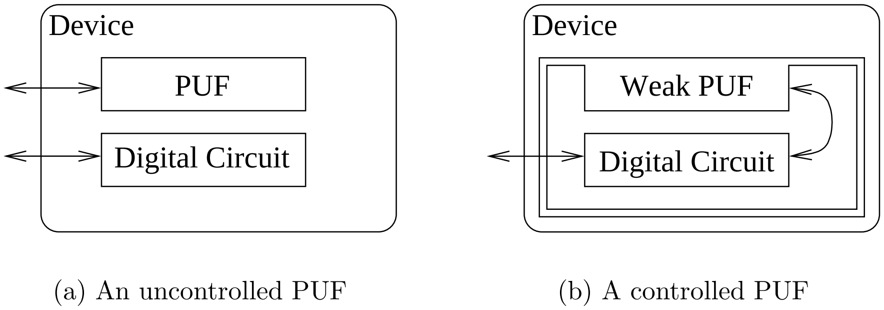
2.1.2 Controlled PUFs

Definition 2 *A PUF is said to be Controlled if it can only be accessed via an algorithm that is physically linked to the PUF in an inseparable way (i.e., any attempt to circumvent the algorithm will lead to the destruction of the PUF). It is then called a Controlled PUF (CPUF). In particular the control algorithm can restrict the challenges that are presented to the PUF, limit the information about responses that is given to the outside world, and/or implement some functionality that is to be authenticated by the PUF.*

The definition of control is quite strong. In practice, linking the PUF to the algorithm in an inseparable way is not trivial. However, we believe that it is easier to do than to link a conventional secret key to an algorithm in an inseparable way, which is what classical devices such as smartcards attempt.

Control is the fundamental idea that allows PUFs to go beyond simple identification applications such as the one presented in Section 2.2. Control plays two major parts. In Chapter 4 we will see that control can protect a weak PUF from the outside world, making it into a strong PUF. Moreover, in the assumptions of the control definition, the control logic is resistant to physical tampering, which means that we can embed useful functionality that needs to be protected into the control logic.

In practice the PUF device will be designed so that the control logic is protected by the fragile physical system that the PUF is based on. Any attempt to tamper with the former will damage the latter. Figure 2-1 illustrates how the PUF is intertwined with its control logic in a CPUF.



(a) An uncontrolled PUF

(b) A controlled PUF

Figure 2-1: Control logic must be protected from tampering

2.1.3 Manufacturer Resistant PUFs

Definition 3 *A type of PUF is said to be Manufacturer Resistant if it is technically infeasible to produce two identical PUFs of this type given only a polynomial amount of resources.*

Manufacturer resistance is an interesting property; it implies unclonability and greatly reduces the level of trust that must be placed in the manufacturer of the device. Our way of making PUFs manufacturer resistant is to measure parameters of a physical system that are the result of process variation beyond the control of the manufacturer.

2.2 Simple Keycard Application

The simplest application for PUFs is to make unclonable key cards. This application was first described in [21]. These keycards would be as hard to clone as the silicon PUFs that we present in Section 3.2. Without unclonability, a keycard that has been lost or lent should be assumed to be compromised, even if it is later retrieved.

We describe a protocol for keycards that allows secure identification, in which some person with access to the card can use it to gain access to a protected resource. The typical scenario is that of a person with a key card presenting it to a reader at a locked door. The reader can connect via a secure channel to a remote, trusted server. The server has previously established a private list of randomly chosen CRPs with the card. When the person presents the card to the reader, it contacts the server using the secure channel, and the server replies with the challenge of a randomly chosen CRP in its list. The reader forwards the challenge to the card, which returns the corresponding response. The reader forwards the response back to the server via the secure channel. The server checks that the response matches what it expected, and, if it does, sends an acknowledgment to the reader. The reader then unlocks the door, allowing the user to pass.

The protocol works because, to clone a keycard, the adversary would have to either guess which CRPs are in the database, or build a database of his own that covers a large enough portion of the CRPs that he has a significant chance of knowing the CRP that the reader will ask. For a big enough PUF, both tasks are infeasible.

Because the server's database contains a small number of CRPs, the method only works if each challenge is be used only once. Otherwise, a replay attack is possible. An attacker harvests some challenges from the reader by trying to get access with a fake card. At a later time he gets access to the real card, and asks it for the responses to the challenges that he has harvested. Finally, he goes back to the reader, and keeps trying to get access until he is asked one of the challenges for which he knows the response. Since the reader only contains a small number of CRPs, the adversary has a significant chance that one of the ones he harvested will get reused. The protocol worked only as long as an attacker was unable to guess which of the unmanageable multitude of challenges would be used. As soon as the attacker was able to find out that some challenges were more likely, he could make himself a partial clone of the keycard, and get unauthorized access. Because each CRP can only be used once, some people like calling this protocol a one-time-pad protocol.

The fact that CRPs can only be used once leads to the possibility of denial of service attacks in which an adversary tries to gain access until the server has run out of CRPs. One way of mitigating these attacks would be to use a classical challenge-response protocol to authenticate that the card knows some digital secret, and only then using the PUF authentication. That way, the denial of service attack is as hard as breaking the classical keycard, while gaining unauthorized access additionally requires breaking the PUF authentication protocol.

There remains the problem of storing a large number of CRPs for each keycard or renewing CRPs once the database runs low. There doesn't seem to be a cheap way of doing either task, which shows how limited this protocol is. The protocol is interesting nevertheless as it is the only PUF protocol that doesn't require control. We shall explain this limitation of uncontrolled PUFs in Section 5.2.1.

The keycard only solves part of the problem of getting a door to open only for authorized users. Other aspects include making the door sturdy enough to prevent forcible entry, making the server and reader secure, and finding ways such as biometrics or Personal Identification Numbers (PIN) for the user to identify herself to her keycard. Indeed, the goal is usually to give access to a person, and not to any bearer of a token such as a keycard.

2.3 Threat Model

The reason for using PUFs rather than classical methods is to try to improve physical security. In this section, we try to quantify the improvement by considering a variety of threat models. First we will consider the abilities the attacker might have, then the goal he is trying to achieve. Finally we will consider a few typical attack scenarios.

2.3.1 Attack Models

There is a wide range of different attack models to choose from:

Passive vs. Active

An attacker can simply observe a device (passive), or he can intercept, modify or create signals of his own (active). For any reasonable level of security, an active attacker must be assumed.

Remote vs. Physical Access

Classical methods are perfectly suited to defeating attackers that only have remote access to a device. To be useful, PUFs must do just as well against remote attackers.

The real potential that PUFs have for improvement is against attackers with physical access. There is a whole range of physical attackers to consider. Non-invasive ones will simply observe information leaking from the device without opening it. Invasive attacks will actually open the device. Destructive attacks will damage the device. Non-destructive ones won't leave any trace of the attacker's work, allowing him to return the device to its owner without arousing suspicion.

With physical attacks, a level of technology has to be assumed: there doesn't seem to be much we can do against an adversary who can take the device apart an atom at a time, and then put it back together again. Consequently we won't be able to make any absolute statements about PUF security against physical attacks. Both financial resources and state of the art technology must be considered to determine an attacker's physical ability.

Openness of Design

Detailed information about the design of the PUF device can be available to the attacker or not. Kerckhoffs' [14] principle says that full knowledge of the design by the attacker should be assumed and only key information should be assumed private.

In the case of PUFs, the key information is typically the process variation that was present during device fabrication. For manufacturer resistant PUFs, that variation can't be controlled and/or known by the manufacturer. For PUFs that aren't manufacturer resistant, the Manufacturer might have some knowledge of or control over that information.

Computational Ability

The attacker can have bounded (e.g., polynomial) or unbounded computation at his disposal. Nevertheless, even an attacker with unbounded computation will be limited in the number of queries he can make from the PUF as each query must be run through the single instance of that PUF in existence.

Online or Offline

If the attacker can attempt an attack while the PUF is in use, we call him an online attacker. If has access to the device only when it is idle, he is an offline attacker.

An online attacker might be the owner of a PUF equipped set-top box. An offline attacker would be trying to clone the PUF equipped credit card that you left on your desk during a lunch break.

2.3.2 Attacker Success

Possible goals the attacker may have are listed in approximate order of decreasing difficulty:

Attacker Clones the PUF

The attacker produces a device that is indistinguishable from the original. The word indistinguishable can take on a range of meanings. For example, in copying a smartcard, the attacker might need to produce a card that looks just like the owner's original card, and that passes for the original in the card reader. When cloning a phone card, however, the clone only has to look like the original to the card reader.

Attacker Produces a Partial Clone of the PUF

In this case, the clone isn't perfect, but is good enough for the clone to have a non- negligible probability of passing for the original.

Attacker Gets the Response to a Chosen Challenge

Given any challenge, the attacker is able to get the corresponding response. This attack, like those that follow, is only useful in the context of CPUFs, as uncontrolled PUFs are willing to give the response to any challenge to whoever asks.

Attacker Tampers with an Operation

Without being detected, the attacker is able to make a CPUF device perform an operation incorrectly. The tampering can be more or less deliberate (i.e., the attacker chooses how exactly the operation is carried out), which leads to a whole range of degrees of success.

Attacker Eavesdrops on an Operation

Without being detected, the attacker is able to read information that should have been withheld from him.

Attacker Gets the Response to Some Challenge

The attacker manages to find out any CRP. This attack is actually not very interesting for us except in Chapter 6. Indeed, we would like anybody to be able establish trusted relations with a CPUF device, and this will involve knowing CRPs.

Manufacturer Produces two Identical Devices

This would violate the manufacturer resistance assumption. We can also imagine that the manufacturer would be able to produce two nearly identical devices that could pass for one another with non-negligible probability.

This security breach only applies for manufacturer resistant PUFs, it is independent in strength from all the attacker goals except cloning.

2.3.3 Typical Attack Scenarios

We now describe some typical attack scenarios that we will be trying to prevent in the rest of this thesis.

Brute-Force Attacks

Naturally, there are a number of brute force attacks that attempt to clone a PUF. An adversary can attempt to produce so many PUFs that two will be identical. This is implausible, as the number of PUFs to produce would be exponential in the number of physical parameters that contribute to the PUF. Moreover it is not clear how to check if two PUFs are identical.

Alternatively, an adversary can attempt to get all the CRPs by querying the PUF. This also fails as the number of CRPs to get is exponential in the PUF input size. The limitation is more severe than in the previous case because all the CRPs must be extracted from a single device that has a limited speed.

Model Building

An attacker could produce a generic model of a given class of PUF devices, and then use a relatively small number of CRPs to find the parameters in his model for a specific instance of the device. We can hope to make this attack difficult for a computationally bounded attacker.

Direct Measurement

An adversary could open the PUF device and attempt to measure physical parameters of the PUF. Those parameters could then be used to manufacture a clone (not for manufacturer resistant PUFs) of the device, or could be used as parameters for a model of the device. We will often assume that this attack is infeasible as the attacker has to open the device without damaging the parameters that he wishes to measure.

Information Leakage

Most protocols that a CPUF could be involved in require that some information being processed in the device remain private. Unfortunately devices tend to leak information in many ways. Power analysis, in particular, has attracted considerable attention in recent years [15], but it is not alone [18, 2]. Therefore CPUFs must be designed with the possibility of information leaks in mind if they involve any manipulation of private data.

Causing a CPUF to Misbehave

In the case of a CPUF, the adversary can attack the control logic that surrounds the PUF. He could do so to extract information that should have remained hidden, or generally make the device operate in an unexpected way without arousing suspicion.

This type of attack is present with classical methods [3]. Clock glitches, voltage glitches, temperature extremes or radiation can all be used to cause misbehavior. The classical response is to equip the device with sensors that detect these conditions, or regulate parameters internally. These attacks and defenses still apply in the case of CPUFs.

PUFs can be used advantageously in the case of invasive attacks that try to cause the device to misbehave by tampering with its internal circuitry. Indeed, if the PUF is intertwined with the critical circuitry, attempts to modify the circuitry are likely to damage the PUF causing the tampering to be detected. This intertwining is to be compared with classical methods [25, 23] that surround the device with a mesh of intrusion sensors causing the device to clear its memory when an intrusion is detected. These intrusion sensors must be turned on permanently to prevent an attacker from opening the device while it is off and doctoring the tamper detection circuitry. With PUFs, there is no need for constant active monitoring; the adversary will damage the PUF — which is the device's secret information — whether it is on or off.

Open-Once Attacks

A final interesting model that we will consider is the *open once* model. In this model, the attacker cannot exploit information leakage, and he destroys the PUF as soon as he opens the device. However, it is assumed that the attacker can open the device at any time and get full access to the digital state of the device (he can read and tamper at will) at any single time. In doing so, though, he destroys the PUF. Essentially we have a black box containing a PUF and some other circuitry. The PUF works as long as the box is closed, and once the box has been opened so that we can play with the internal mechanisms, it can't be re-closed.

This model might appear contrived. Nevertheless, it leads to interesting results and is plausible if information leakage can be prevented, and the PUF can be caused to break as soon as the device is opened.

2.4 PUF Architecture

The definition that was given of a PUF is concerned mainly with functionality. We shall now see an overview of different ways of implementing a PUF.

2.4.1 Digital PUFs

We can contemplate making a PUF by building a circuit that combines its input with a secret using a pseudorandom function, and placing it in an impenetrable vault as in Figure 2-2. This is an extreme case of the PUF as it is built with classical digital primitives. A digital PUF has all the vulnerabilities of any digital circuit, as well as the complications of using PUF protocols. Clearly, there is no point in using digital PUFs.

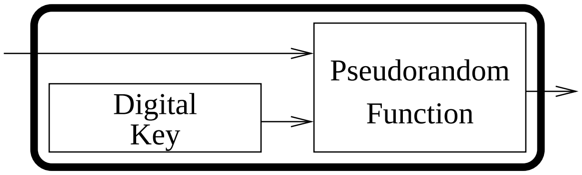


Figure 2-2: Architecture of a digital PUF

2.4.2 Physically Obfuscated Keys

If we eliminate the vault from the digital PUF, then the digital key that is inside can be revealed to an adversary that performs an invasive attack. Therefore, we can try extract the key from a complex physical system that is harder for an invasive adversary to comprehend as in Figure 2-3. By using the complex physical system as a shield for the digital part of this device (to prevent an attacker from opening it and inserting probes to get the key), it might be possible to make a PUF. We will consider this type of idea in Chapter 6. Nevertheless, it would be nice not to have a single digital value that, if probed, reveals all the device's secrets.

2.4.3 Analog PUFs

In analog PUFs, we try to put as much functionality as possible into a complex physical system. That system takes a challenge and directly produces a response.

In practice, it is hard to find a physical system that has the characteristics we desire. The output won't be completely unpredictable, and because of measurement noise, the output for a given challenge is likely not to be completely reliable. We can say that we have a weak

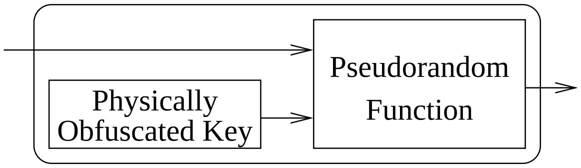


Figure 2-3: Architecture of a physically obfuscated PUF

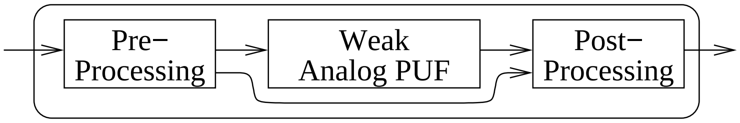


Figure 2-4: Architecture of an analog PUF

PUF. To solve these problems, some digital pre- and post-processing has to be added to the physical system as is shown in Figure 2-4.

The upcoming chapter will look at candidate physical systems for an weak analog PUF. Then in Chapter 4 we will look at the digital processing that is needed to strengthen the PUF.

Chapter 3

Analog Physical Random Functions

In this chapter, we will discuss ways to produce a weak analog PUF. This is the basic building block that we strengthen into a full-fledged PUF in Chapter 4. The weaknesses that we are willing to tolerate in this chapter have to do with PUFs that have noisy output and for which modeling attacks might be possible.

We will briefly consider PUFs based on optical systems, then move on to the silicon systems, which have been our main focus. This is in no means an exhaustive presentation of ways to make weak analog PUFs.

3.1 Optical Approaches

3.1.1 Physical One-Way Functions

Physical One-Way Functions [21] are a simple example of analog PUFs (see Figure 3-1). The physical system is a wafer of bubble-filled transparent epoxy. The PUF behavior is obtained by shining a laser beam through the wafer from a location that is precisely defined by the challenge. The laser beam interacts in a complex way with the bubbles in the wafer, and a speckle pattern is observed on the other side of the wafer. This speckle pattern is a function of the exact position of the bubbles and of the laser, and is the PUF's response.

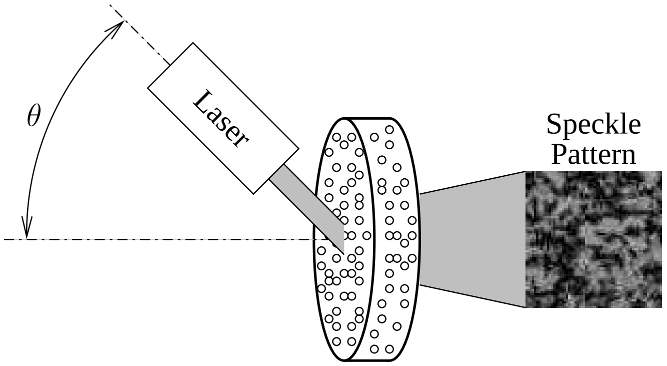


Figure 3-1: An optical PUF

In [21], particular emphasis is put on the fact that the wafer that is being measured is a 3-dimensional system. This is very good because compared with a 2-dimensional system, it increases the amount of information in the system, and makes direct measurement harder. This is a significant advantage over the silicon PUFs that we will consider in Section 3.2 that are essentially 2-dimensional.

3.1.2 Analog or Digital Interface

Unlike all the other PUFs that we will consider, the Physical One-Way Function is made to be used with an analog interface: the wafer is embedded in a plastic card and all the expensive optical equipment (laser, speckle detector) is part of the reader. This is interesting because an adversary who wants to clone a wafer by using a model building method has a much harder task. Indeed, not only does he have to be able to predict the speckle pattern for a given laser position, he also has to determine the laser position, and simulate the speckle pattern on the speckle detector in the time interval that is expected by the reader.

We have decided not to pursue analog interfaces as we are more interested in applications where PUF devices are used remotely. That implies that the measurement equipment is just as vulnerable to physical attack as the physical system it is measuring, and the advantages of having an analog interface disappear. Therefore, we prefer to include the measurement equipment into the PUF, and provide a digital interface (i.e., you digitally input the laser's position, and read out a digital representation of the speckle pattern).

A major advantage of incorporating the measurement equipment in the PUF is that there is a reduced need for calibration. If many sensors in many readers have to be able to measure a physical system, then they must all be identically calibrated, whereas if the system is always measured with the same sensor, calibration is unnecessary.

Finally, full fledged CPUFs are only possible if the measurement equipment is part of the PUF. Indeed, a CPUF should be able to restrict challenges and responses it receives, which is not the case if it does not incorporate the measurement equipment.

Of course, for the digital interface method to be cost effective, the sensor equipment must be cheap. That is certainly not the case for Physical One-Way Functions where precision mechanisms are necessary to position the laser source correctly.

3.1.3 Optical CPUF

The Physical One-Way Function described above can, in principle, be made into a potentially cost effective CPUF. The key idea is to integrate the light source and light sensors on a chip that is embedded in an irregular transparent medium like the epoxy wafer that we have been considering, and surrounded by reflecting material. The setup is depicted in Figure 3-2.

Instead of mechanically moving a laser source in accordance with the challenge, many laser diodes are present on the chip, and depending on the challenge, a combination of them is be turned on. Since the different laser diodes are not coherent with each other, a speckle pattern with reduced contrast results. To avoid excessive loss of contrast, only a few sources should be turned on at any given time.[1] Moreover, a non-linear optical medium should be

[1] If the different sources could be made coherent with each other, this constraint would no longer hold. The

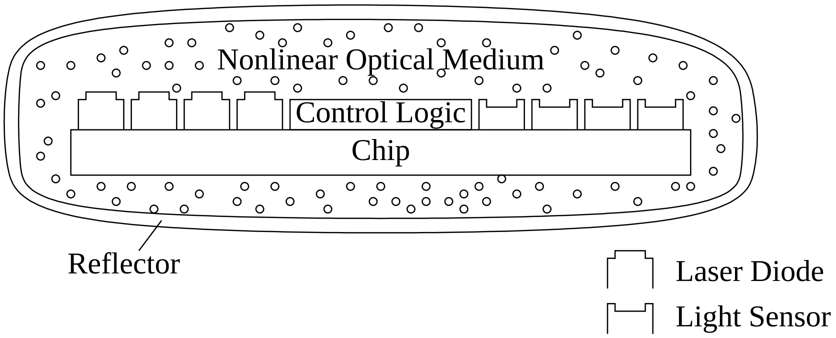


Figure 3-2: An optical CPUF

used so that the speckle pattern isn't simply the sum of the patterns that would result from each source being turned on individually.

With this setup, a control algorithm could be embedded on the chip. The algorithm would be protected from tampering by the optical medium, which would get damaged by any invasive physical attack. Of course information leaks and non-invasive tampering still have to be dealt with.

For now we have not seriously considered implementing this optical CPUF, but it bears some resemblance to preexisting intrusion detection systems [1]. It is interesting for many reasons: it is based on a 3-dimensional physical system, the control logic is clearly intertwined with the PUF, it is manufacturer resistant, and there is hope that there will be better temperature stability than with purely silicon PUFs. On the downside, the bubble-filled epoxy might be a problem for heat dissipation and the processes that enable optical components to be present on-chip are still costly.

3.2 Silicon Approaches

Instead of combining optics and microelectronics to produce a PUF, we have focused on ways to produce PUFs that only use cheap and extensively studied silicon technology. This section describes the approaches that we have considered to make a PUF out of an integrated circuit (IC).

Silicon PUFs try to harness the variations that are present when chips are fabricated. These variations and their sources are under constant research [6] as they determine the minimum feature size on ICs. In each new process, chip manufacturers do their utmost to reduce variation. Yet, with each generation, the relative variation increases (Chapter 14 of [8]). This observation shows us that the process variation in ICs might be an excellent starting point for manufacturer resistant PUFs since this variation exists despite the manufacturing world's best effort.

Using process variation to provide chips with a unique identifier has already been done [17] by measuring drain currents in transistors. However, this method reduces the variation to

number of challenges would then grow exponentially with the number of sources, rather than polynomially.

a vector of bits, that is easy to clone by embedding those bits same bits into a device that only pretends to measure physical parameters. The major question that we must answer is how to go beyond simple identification, and use process variation to produce a PUF.

3.2.1 Overclocking

For the most part, circuits on ICs are designed in the synchronous logic model: a huge combinational circuit maps a vector of inputs to a vector of outputs, and a bank of registers connects a subset of the outputs to a subset of the inputs (see Figure 3-3). The whole circuit is synchronized on a clock that ticks at regular intervals. Each time the clock ticks the registers instantly copy their input to their output. The combinational logic sees its new input, and propagates it through its logic gates, to compute the new output. While the logic is computing, the output can go through some glitching as signals propagate through the logic at different speeds. Eventually the output settles and the circuit is ready for the next clock tick.

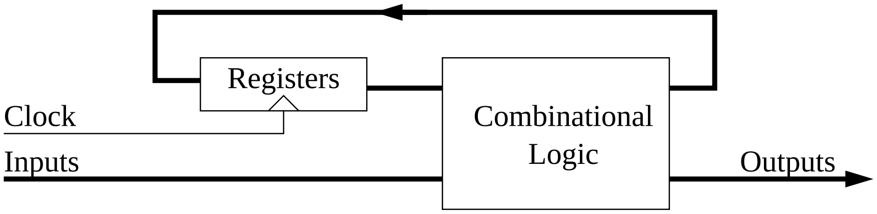


Figure 3-3: A general synchronous logic circuit

As long as the clock frequency is low enough, the output always has time to settle, and the circuit behaves as expected.[2] Fabrication variation only shows up in the details of how signals propagate through the combinational logic. Once the clock signal arrives, all functional chips will have stabilized at the same output value.

This simple picture has been a key contributor to the successful scaling of electronic devices from thousand transistor circuits to hundred-million transistor circuits. Simply by selecting a clock frequency that is low enough, the effects of all the manufacturing variations can be wiped out. However, if the clock frequency is increased, the clock might tick when the output of the combinational logic is still settling, and the results become chip dependent.

This discussion inspires a very simple type of PUF implementation. A challenge is a whole vector of inputs to the combinational logic and a delay. To query the PUF, the circuit is brought into a state in which it will, at the next clock cycle, have the challenge input going into the combinational logic. At the clock tick, that signal starts propagating through the combinational logic. Instead of being sent after the normal clock period, the next tick is sent after the delay that is contained in the challenge. The vector that is captured by the registers during that tick is taken to be the PUF's response. Essentially, the idea is to tell chips apart by looking at their response to momentary overclocking.

[2] Of course, there are extra constraints in real circuits because registers have more elaborate timing problems, which are irrelevant here.

In principle, this method could be combined with the methods from Chapter 4 to produce a PUF. In practice, there are many difficulties:

- There is a huge number of challenges for which the response will not depend on the chip at all, or will depend very little on the chip (for example, all challenges for which the delay is greater than the normal clock period). For simple circuits, most challenges will be in this case. That means that a method for finding significant challenges must be found, or else the chip must be queried a huge number of times to get enough identifying information out of it (see Section 4.2).

  One way of selecting good challenges would be, to take an arbitrary input vector for the challenge, and look at how the output depends on the delay that we put in the challenge. We pick a delay that is close to a transition of the output. Such challenges would be more likely to vary across chips.

- For complex circuits such as a multiplier, the output vector might vary very rapidly while settling. In order to capture the response consistently, the output vector must be stable for a small period of time around the clock tick called the hold time. If that minimum time is rarely met in the output vector's settling, then few challenges will consistently output the same response. Instead of measuring the device, we would be measuring noise.

- Depending on environmental parameters such as temperature, the settling of the output vector might not occur in the same way. It is not clear how this PUF scheme would be able to recognize a chip that was characterized at a different ambient temperature than the one at which it is used.

- The clock is an extremely sensitive part in a circuit, especially as the circuit size and speed increase. Inserting a precisely timed glitch in its normal operation as this PUF implementation requires is a difficult task.

- The way the combinational logic settles might depend on the previous input as well as the current one, so the previous input would also have to be included in the challenge.

- For complex sequential circuits (in which many outputs of the combinational logic are fed through registers), it is not always possible to bring the circuit into an arbitrary state. Even when it is possible, determining how to do so is very difficult. Therefore, instead of defining the input vector in the challenge, one would need to give a sequence of steps that would bring the circuit into the appropriate state for the PUF measurement to be made.

Clearly, there are a number of problems that arise from this simple PUF implementation. None of them are insurmountable, but so far this doesn't seem like a practical way to make a PUF, though it would have been nice to have a method to make a PUF from an arbitrary synchronous circuit.

3.2.2 Genetic Algorithm

Some interesting work [26] involving FPGAs and genetic algorithms shows that it is possible to evolve a circuit to distinguish between an input that oscillates at $1kHz$ and an input that oscillates at $10kHz$. The interesting part for us, is that the evolved circuit, when loaded into a different FPGA, no longer works. Moreover, when we look at the evolved circuit, it is not at all apparent how exactly it works.

This experiment confirms once again that process variation does make chips measurably different, and also shows a way of producing circuits that are hard to understand, and that tell chips apart. We can in theory base a PUF on this observation. A challenge is an FPGA circuit, and the response is the output of that circuit when it is loaded into the FPGA.

As with the overclocking method, many challenges will produce responses that do not depend on the chip, while others will produce erratic responses. A genetic algorithm can be used to find challenges that meet our needs.

Environmental parameters are also likely to change the responses as is observed in [26], though it might be possible to vary environmental parameters during training to find challenges that are sufficiently robust to environmental parameters such as temperature [27].

Overall this method is neat, and shows that PUFs can be made. Unfortunately, it is impractical because the genetic algorithm takes a prohibitive time to run.

3.2.3 Delay Measurement

So far we have considered two ways of making a Silicon PUF. Both methods were very general, but both had similar problems: it is hard to find challenge-response pairs that actually differ from chip to chip, and it is hard to get reliable operation, particularly in presence of environmental variation (such as changes in temperature). We will now turn to a method that is less general but that we have better control over. The basic idea is to measure delays in a specially designed circuit to produce a PUF.

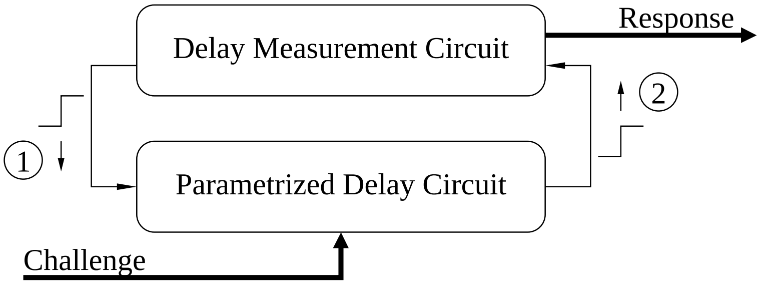


Figure 3-4: Measuring delays to get a PUF

The general architecture of a delay-based silicon PUF is shown in Figure 3-4. The main element is a parametrized delay circuit. When a rising edge is applied to the input of the delay circuit (1), it propagates through the delay circuit. After some amount of time that is a function of the challenge that is applied to the circuit, the rising edge arrives at the circuit's output (2). In the PUF, the delay circuit is coupled to a delay measurement circuit that generates the rising edge, and measures the time until it emerges from the delay circuit. It outputs the delay as the response. As long as the delay circuit is complex enough, we have a good PUF candidate.

Many of the drawbacks that existed with previous circuits have now been solved. Any challenge now gives a response that changes from chip to chip, so we no longer need to look for challenges that stimulate some chip dependent behavior. Since the delay circuit is designed so that one rising edge at the input leads to a single rising edge at the output, the quantity that we are measuring is well defined; as long as the delay circuit is designed so that its delay is the same each time it is stimulated, we will be able to get consistent measurements.

Environmental Parameter Compensation

The problem that remains is robustness to environmental variations. Fortunately, it turns out (sections 3.3 and 3.4) that when temperature changes, delays of similar design vary proportionally to each other. Therefore, by measuring two different delays and dividing them by one another it is possible to get a value that is a lot less dependent on environmental parameters such as power supply voltage and temperature than the values that are being divided. We call ratios of two delays *compensated measurements*, and the operation of dividing delays by each other *compensation*.

We now have a viable silicon PUF candidate. In the next sections we go into the details of the delay measurement circuit and the delay circuit so that we have a complete story on how to build delay-based silicon PUFs.

3.2.4 Measuring a Delay

Self-Oscillating Loops

The main method that we have used to measure delays is to use the delay circuit to build a self-oscillating loop. We do this by inverting the output of the delay circuit and feeding it back into the delay circuit's input. This circuit spontaneously oscillates with a period approximately two times greater than the delay to be measured.[3] The period of the loop can easily be determined by counting the number of oscillations in a given amount of time as measured by some reference clock. Because of compensation, the reference clock doesn't even need to be calibrated, it need only be stable over short time-scales.

Figure 3-5 shows the details of the self-oscillating loop circuit. The `Oscillating` signal is used to turn the loop on or off. The output of the loop is synchronized to the reference clock by using two registers. Another register and an and gate are used to detect rising edges. The rising edges are then counted, as long as the `Counting` signal is high. To perform the measurement, the loop is turned on, the counter is reset using the `Reset` signal, `Counting` is put high for a predefined amount of time, and finally a frequency is read from the output

[3] Since all we care about is getting a consistent measurement out of the chip, the fact that the loop period is only approximately twice the delay of the delay circuit doesn't bother us.

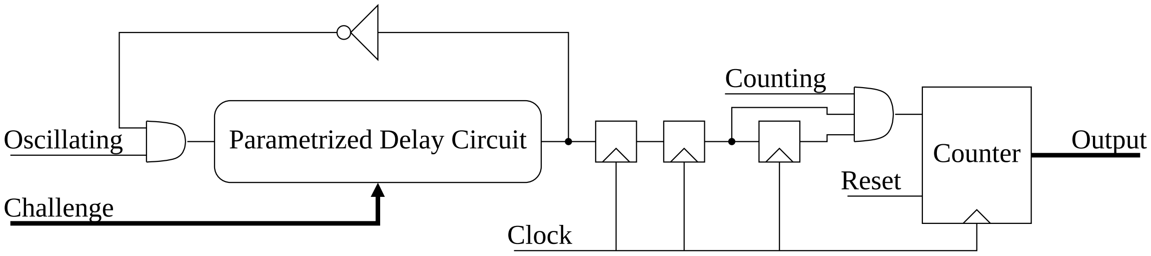


Figure 3-5: A self-oscillating loop to measure a delay

of the counter.[4] Of course the clock must be at least two times faster than the loop for this circuit to function.

A few constraints on the delay circuit must be maintained for self-oscillating loops to work. Indeed, the edge that enters the delay circuit might get split into many competing edges inside the circuit. One of those edges will eventually cause the output of the delay circuit to change, which will almost immediately cause an opposite polarity edge to be applied to the input of the delay circuit. If some edges from the previous time through the loop are still present in the delay element, they could change the delay that is seen by the new edge. With a badly designed delay circuit, this could lead to chaotic variations in the delay, and unreliable responses.

This method leads to very precise measurements. Unfortunately, to get measurements with precisions of tens parts per million, hundreds of thousands of clock cycles must elapse during the measurement. Measuring delays is consequently a slow affair, which gets even longer when we start using vectorization (see Section 4.2).

Arbiters

To make delay measurement faster, we are trying to do direct delay measurements, which only run a single edge through the delay circuit. This is not a trivial task given that in our experiments, we have to measure the delays with precisions on the order of tens of picoseconds. An added advantage of not using an oscillating circuit, is that stable frequencies are particularly easy for an adversary to look for when he is trying to probe a circuit. This weakness could allow an adversary to bypass a CPUF's post-processing steps, such as the output random function (see Section 4.3).

Daihyun Lim is currently running experiments in which, instead of measuring two delays and taking their ratio, two delays are simply compared. Two (possibly intermingled) delay circuits are excited simultaneously. An arbiter circuit at the output determines which rising edge arrived first and sets its output to 0 or 1 depending on the winner. Figure 3-6 shows an arbiter-based circuit.

Preliminary results show that the method works. The disadvantage of the method is that if for some challenge, one delay is consistently greater than another across chips, then

[4] In fact, because the reference clock period must be less than half the loop period, it is better, for maximum precision, to count the number of clock ticks in a predefined number of oscillations of the loop rather than the number of oscillations of the loop in a certain number of clock ticks.

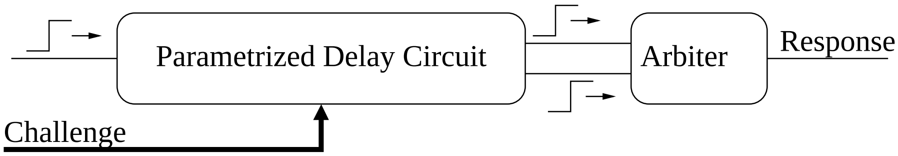


Figure 3-6: An arbiter-based PUF

the response will always be the same for that challenge. Fortunately, each measurement can be performed extremely fast, so it is possible to try many challenges in the time it would take to measure a single delay with a self-oscillating loop. In one experiment we have run, around 2 % of challenges have a response that changes between chips. This figure would be sufficient to make the arbiter circuit faster than the self-oscillating loop circuit despite having to try more challenges. With more control over delay circuit layout, we hope to get much more than 2 % of useful challenges.

3.2.5 Designing a Delay Circuit

Choosing the right delay circuit is the key to making silicon PUFs work. The delay circuit must be designed to have a consistent delay and be resistant to model building attacks.

To make model building difficult, it is tempting to try to reproduce the types of structures that appear in digital pseudo-random functions such as SHA1 [20] or MD5 [22], to make the relation between individual physical parameters and global delay hard to invert. Unfortunately, those functions are designed to have a cascade effect: if a single bit is changed at the input, one can expect half the output bits to change as a result. This chaotic behavior is contradictory with what we need to get reliable delay operation. With a chaotic analog circuit, any small fluctuation in the physical parameters would completely change the circuit output, making it useless for identification purposes (it might be usable as a random number generator, though).

Therefore, we will use the opposite approach. We will start from a circuit that gives reliable outputs, study it, and propose ways of making it harder to model.

Our basic delay circuit is made up of a string of switch blocks. A switch block has two inputs, two outputs and a control bit. The value of the control bit determines if the inputs go straight through to the outputs, or if they are exchanged, as in Figure 3-7. The input of the delay circuit goes to both inputs of the first switch block. One of the outputs of the last switch block is used as the output of the delay circuit, the other is ignored (or possibly used as the other input to the arbiter circuit if we are doing direct measurement). Figure 3-8 shows the full circuit.

This circuit is interesting because each challenge selects a different path through the circuit, leading to an exponential number of paths. Of course, these paths are far from being independent of each other. In fact, for a k stage circuit, if we assume an additive delay model — the total delay of a path is the sum of the delays of the elements that make it up — then knowing $4k$ independent paths is sufficient to set up a linear system of equations, which, when inverted, would give all the elementary delays in this circuit.

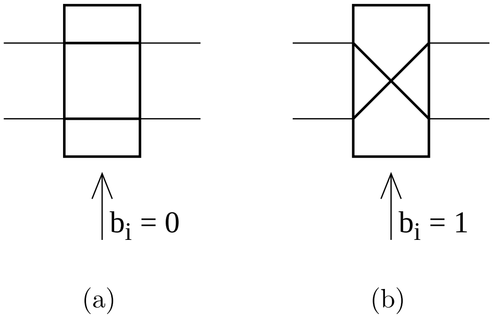


Figure 3-7: When $b_i = 0$ the paths go straight through, when $b_i = 1$, the paths are crossed.

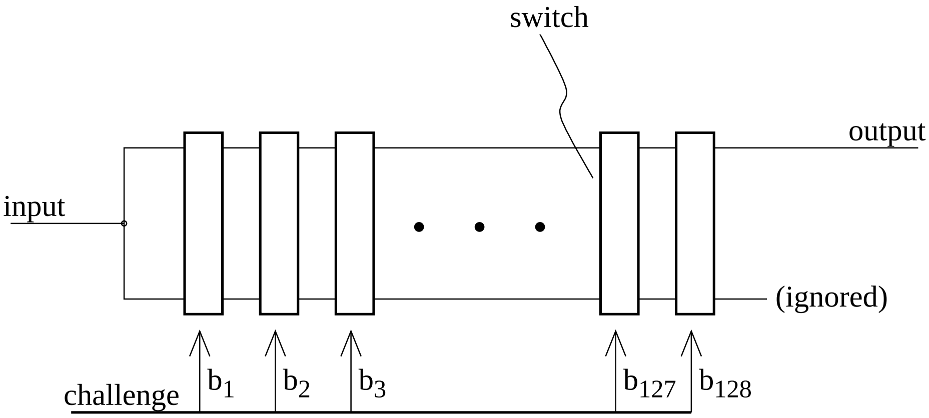


Figure 3-8: The simple delay circuit

Combined with the methods discussed in Chapter 4, this circuit would actually be sufficient for any remote attacker. Nevertheless, just as we choose to use analog PUFs rather than physically obfuscated keys in Section 2.4, it would be nice to make the physical system that the PUF is based on intrinsically harder to model. We shall therefore present a number of tricks that make model building harder.

To date we do not know how much difficulty is added by these tricks if an additive delay model is assumed. What is certain is that, deviations from the additive delay model, and the input pseudorandom function that we introduce in Section 4.1, makes attacks even more difficult.

Variable delay buffers

Variable delay buffers can be used to make additive delay modeling much more difficult. Essentially they are delay elements that have a delay which depends on an auxiliary input. The subtlety is that no glitches must be introduced if the auxiliary input changes while an edge is propagating through the delay element, or else measurement reliability would decrease.

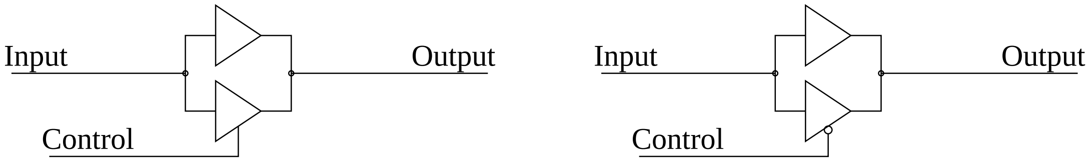


Figure 3-9: Variable delay buffers can be made with a buffer and a tristate buffer. The tristate buffer can be active-high as on the left, or active-low as on the right. Depending on the polarity of the edges and whether the variable delay buffer is active-high or active-low, competing edges will be brought closer together or farther apart.

Variable delay buffers can easily be made by combining a slow buffer (essentially a yes logic block) and a fast tristate buffer (a buffer that can have its output turned off) as in Figure 3-9. If the tristate buffer is off, the delay is long. If the tristate buffer is on, the delay is short. If the tristate buffer changes state while an edge is propagating through the buffer some intermediate delay results.

Since our basic circuit has two edges propagating through it simultaneously, we can use one of the edges as the auxiliary input to a variable delay buffer, and have the other path go through the buffer as is shown in Figure 3-10.

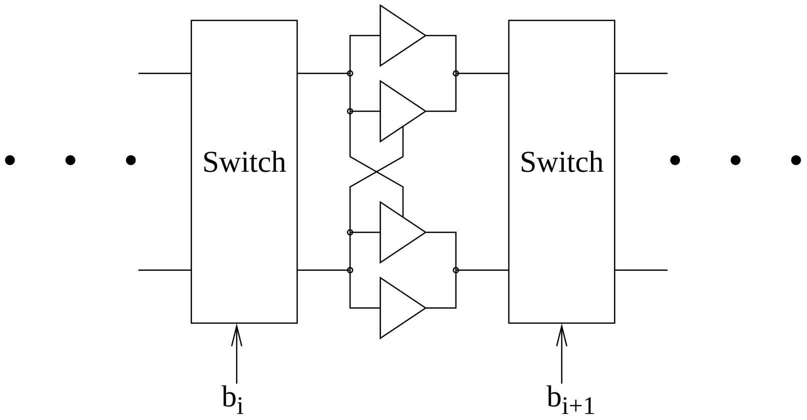


Figure 3-10: Variable delay buffers in a delay circuit

The presence of variable delay buffers makes additive delay modeling much harder as the delays of individual circuit elements are no longer constant. The adversary must make guesses about which state the variable delay buffer is in. In fact the variable delay buffers have a delay that can vary continuously in a certain range which makes the adversary's life even harder. It is interesting to note that variable delay buffers can be used to create non-monotonic behavior: a component in the circuit that becomes faster can make the overall circuit slower.

Great care must be taken when designing a circuit with variable delay buffers. They lead to circuits with complicated behaviors. If misused, a number of things can go wrong. The circuit can become chaotic, and therefore useless. If the variable delay buffers tend to make the leading path get even further ahead than it already is, then the two competing paths will get well separated after just a few circuit stages and the remaining variable delay buffers will be in predictable states.

Overall, variable delay buffers seem like a powerful but potentially dangerous tool. A lot more research could be devoted to exploring delay circuits based on them.

Minimum and Maximum

A somewhat more controlled way of preventing a model building adversary from setting up linear systems of equations is to put max and min operators in the model he is building. For direct delay measurement, this can be done using an and-gate or an or-gate respectively. For self-oscillating loop circuits the task is a little harder as there are both rising and falling edges to take into account, we will see how to handle that case in Section 3.4.1.

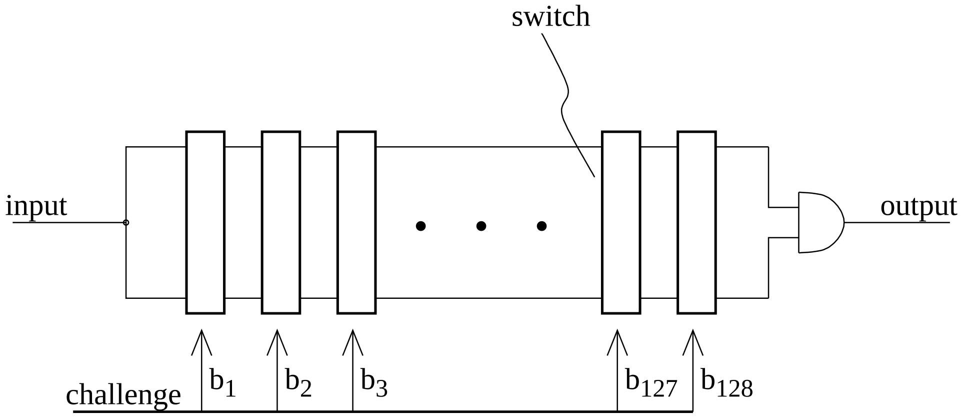


Figure 3-11: Delay circuit with max operator

One way of using these operators to make model building harder is shown in Figure 3-11. A model building attacker doesn't know which path was selected by the and-gate, which forces him to produce two equations for each CRP he knows. An exponential number of possible systems of equations results when many CRPs are combined, unless some pruning strategy is found. For a while [9], our hope was that as long as our delay circuit was laid out in a way that preserved the symmetry between the two paths, and an input random function was used on the challenge, this circuit would be hard to break in the additive delay model. Recent unpublished work by Marten van Dijk shows that such a circuit is vulnerable to statistical attacks, even with an input random function. Perhaps incorporating more max and/or min operators in the circuit would thwart these attacks.

Arbiters

Yet another way of making modeling attacks harder is to have some of the switches be controlled by signals that are generated earlier on in the delay circuit. For example, Figure 3-12 shows how an arbiter can decide, part way through the circuit, which path is fastest so far, and set a switch further down the circuit based on that decision.

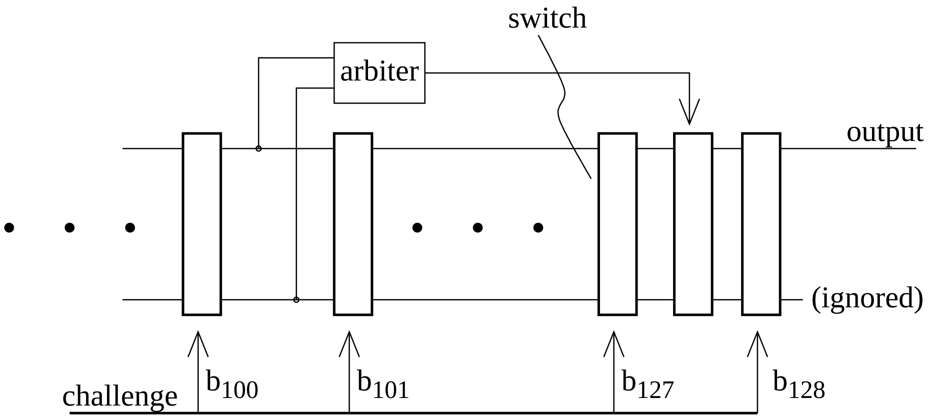


Figure 3-12: Adding internal variables with an arbiter. Note that for simplicity, this figure does not include the logic needed to reset the arbiter between edges.

The attacker's job is harder because he has no way of knowing what the arbiter's decision was. The drawback is that there are borderline cases where the edges arrive at the arbiter almost simultaneously. Those cases will lead to erratic responses. Therefore, the number of arbiters that can be placed in the circuit this way must remain limited.

3.2.6 Intertwining

In a CPUF, an invasive adversary must be prevented from tampering with the control algorithm. Our way of addressing this problem is to design the CPUF so that the control logic is protected by the physical system that the PUF is based on. That way, the adversary is forced to damage the physical system if he tries to tamper. In the case of silicon PUFs, we suggest that the top layer of wires above the control logic be devoted to delay wires, as shown in Figure 3-13. Those delay wires would then act as a shield for the underlying logic.

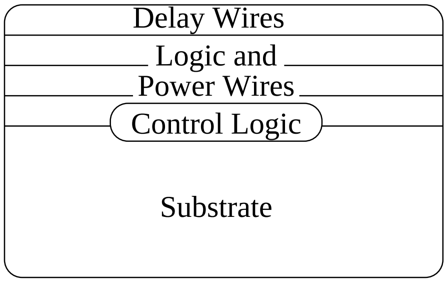


Figure 3-13: Control logic, protected by overlying delay wires

At this time, we do not have the means to test the difficulty of tampering with the control logic while leaving the PUF sufficiently intact. Attacks through the top of the chip are greatly hindered because the adversary must gain access to lower layers of the chip without damaging PUF wires. Unlike attacks against classical circuits that surround the

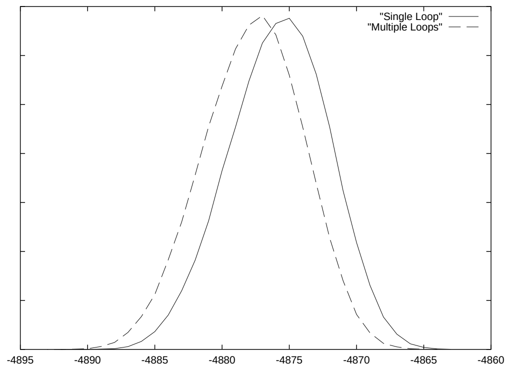


Figure 3-14: In this plot we show how multiple self-oscillating loops on the same IC interfere. The histograms show the frequency of the loop oscillating alone or with the seven other loops on the IC turned on. The frequency shift between these two situations is tiny compared with measurement noise.

chip with meshes, the adversary is detected as soon as the wire delay starts changing; he does not need to cut the wire completely to change the PUF.

Attacks through the bottom of the chip are not taken into account with this approach, but to date, invasive attacks on chips have concentrated on entering through the top. Entering through the bottom of the chip without destroying it seems much more difficult.

If the adversary opens the chip, but does not attempt to penetrate the PUF wires, he can try to make measurements of the signals flowing in the delay wires. Those measurements could help him build a model of the circuit. His task is particularly easy in the case of self-oscillating loop measurements. He simply needs to determine the frequency of the oscillation to know the output of the PUF before any post-processing (see Chapter 4). A possible solutions is to design the chip's packaging so that its removal will sufficiently change the delays of the PUF wires (preferably in an inhomogeneous way to make sure that compensated measurements won't cancel out the change). In the case of direct delay measurement, arranging to have a large number of delay elements in lower levels of the chip, so that the adversary can only get a limited amount of information for his model by direct measurement, is another possibility.

3.3 Experiments

The rest of this chapter describes experiments that were performed using XC2S200 Field Programmable Gate Arrays (FPGAs)[5] on various delay circuits. FPGAs are an example of a high-volume part where the manufacturing process is tuned to produce ICs that are as identical as possible in order to maximize yield and performance. Our experiments indicate that even a highly-optimized manufacturing process designed for predictability has enough variability to enable reliable identification.

[5] The exact components that were used were the XC2S200PQ208-5.

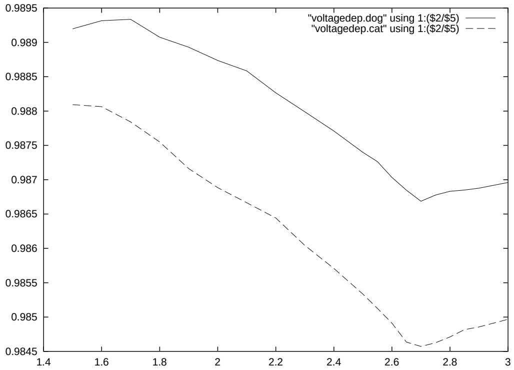


Figure 3-15: This plot shows compensated measurement dependency on power supply voltage. The dependency for 1% changes in supply voltage is small enough for our purposes. Interestingly, by running the FPGAs near the $2.7V$ extremum, it might be possible to further reduce the voltage dependency.

In all our experiments, we compare delays across two or more FPGAs with each FPGA being programmed by exactly the same personality matrix. This means that each FPGA has exactly the same logic circuit, and moreover the circuit is laid out in the exact same locations on each FPGA. Therefore, these FPGAs can be viewed as copies of the same IC.

First we consider a non-parametrized delay circuit, just to see how much variability is present, in Section 3.3.1. Then, in Section 3.3.2 we try to evaluate how valid the additive delay model is, using the simplified circuit in Figure 3-18. Finally, a circuit that might actually provide enough security for a real application is presented in Section 3.4.

3.3.1 Quantifying Inter-Chip Variability

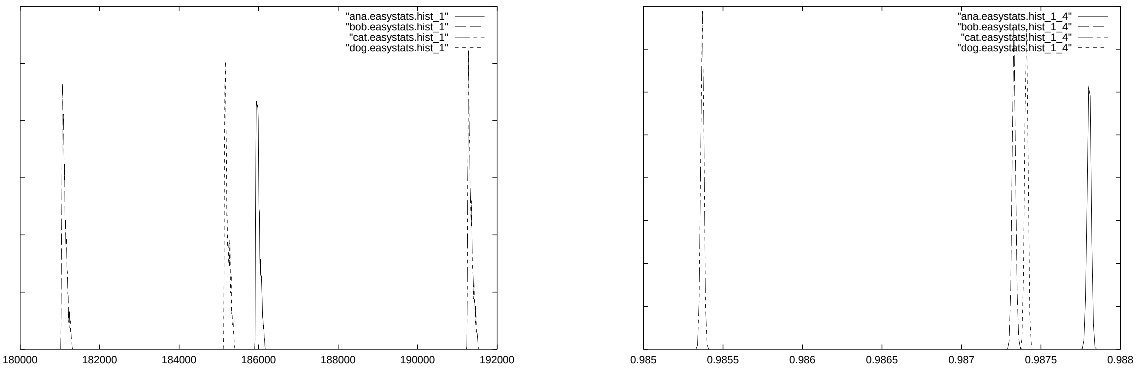
In our first experiment, each FPGA is equipped with 8 self-oscillating loops, the circuit for which is shown in Figure 3-5. Each loop is made up of 32 buffers[6] and an inverter. We determine the frequencies of the loops by measuring the number of oscillations they make during a certain period of time (typically 2^{20} cycles of an external 50 MHz oscillator). The period of the loops is on the order of $60ns$.

We ran various experiments to quantify measurement errors, inter-FPGA variation, variation due to ambient temperature and variation due to power supply voltage variations. To summarize our findings, the following standard deviations are given in parts per million (ppm). A deviation of *n ppm* around a frequency f_0 corresponds to a deviation of $\frac{nf_0}{10^6}$. These deviations correspond to measurement across several FPGAs.

Here are the main results:

1. Consecutive measurements of the same delay produce slightly different results because

[6] In this context, a buffer is simply a logic gate that copies its input to its output with a short delay. They are usually used to amplify signals.



(a) Uncompensated

(b) Compensated

Figure 3-16: These histograms show the relation between measurement error (width of a peak) and inter-FPGA variation (each peak is for a different FPGA), with and without compensation. Clearly information about the FPGA's identity can be extracted from these measurements.

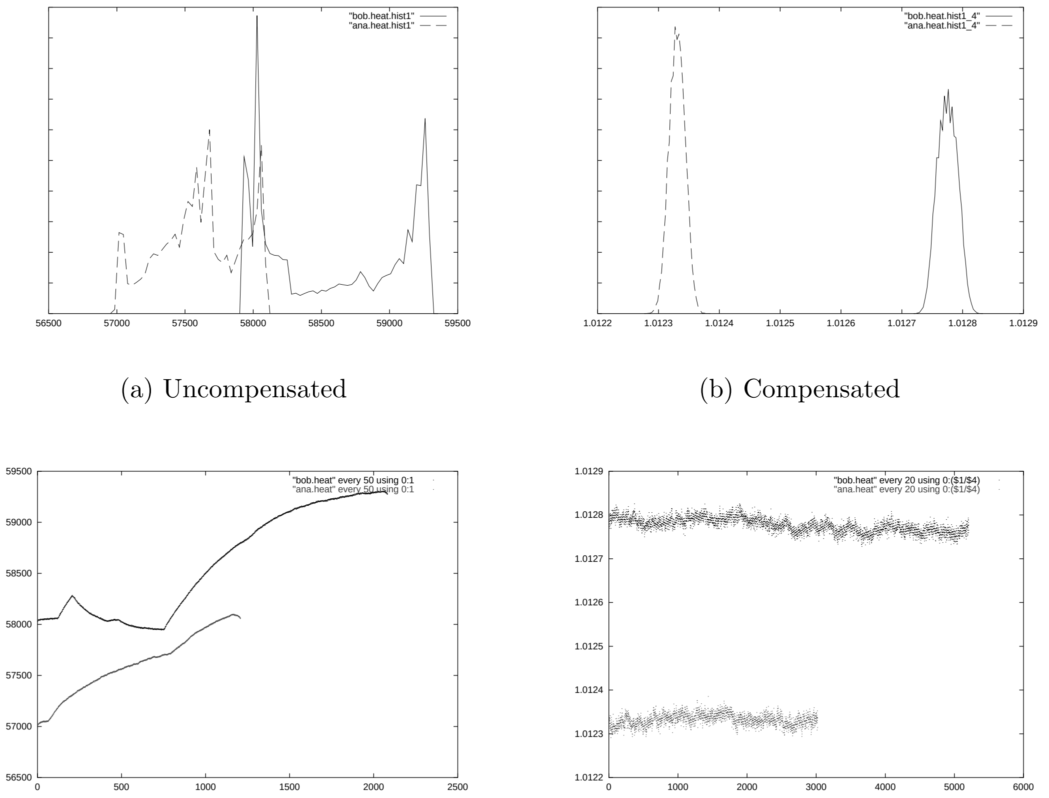
of measurement inaccuracy inherent in the loop circuit. The standard deviation of this measurement error with compensated measurement is 30 *ppm*.

2. The standard deviation in inter-FPGA delays with compensated measurements ranges from 5000 *ppm* to 30000 *ppm* depending on the pair of loops that is used for the measurement. Figure 3-16 shows an example of the relationship between measurement error and inter-FPGA variation for four different FPGAs. Clearly identification information can be extracted from the frequencies of the loops that we are measuring.

3. The frequency of a loop can be influenced by nearby circuitry. To try to evaluate the magnitude of this interference, we compared the frequency of one of the loops when the other loops on the FPGA were turned on or off. The deviation we observed was 10 *ppm*. Figure 3-14 shows the frequency distribution for a loop when the other loops are turned on or off.

4. The variation in frequency when the ambient temperature is varied from 25 to 50 degrees Celsius is 50000 *ppm* for uncompensated measurements. This is sufficient to prevent FPGA identification. Fortunately, with compensation (see Section 3.2.3), this reduces to 100 *ppm*. Figure 3-17 illustrates the temperature dependence with and without compensation.

5. Power supply voltage variations are also well compensated for by our scheme. Around the FPGA's 2.5V operating point, the variation of the compensated measurement with voltage is about $3000 ppm/V$ as shown in Figure 3-15. In practice, external power supply variations can be kept to within 1%, which corresponds to $1\% \times 2.5V \times 3000 ppm/V = 75$ *ppm*. Therefore, commonly available voltage regulators will suffice



(a) Uncompensated

(b) Compensated

(c) Uncompensated

(d) Compensated

Figure 3-17: These graphs show the results of an experiment in which two FPGAs had their ambient temperature vary between 25°C and 50°C. The top plots show the measurement value versus time (in half-second sampling intervals). Note that the two FPGAs did not undergo the same temperature changes at the same time. The bottom plots are histograms of the respective plots on top.

to keep the supply voltage within tolerable bounds. It is interesting to note that the compensated measurement seems to have an extremum around $2.7V$. By running the FPGAs at $2.7V$ instead of $2.5V$ this extremum could be used to further improve the robustness of the measurements.

6. Circuit aging can create variance in measurements carried out over a long period of time. However, the effect of circuit aging is typically significantly less than power supply or temperature variation. Future study will have to check the impact of aging on the measurements.

Given the numbers above, if we take 100 *ppm* as a rough estimate of the noise, and 10000 *ppm* as a rough estimate of the signal, then we have a signal to noise ratio of 100. If the noise distribution was Gaussian (this is not really the case as some parts of the noise

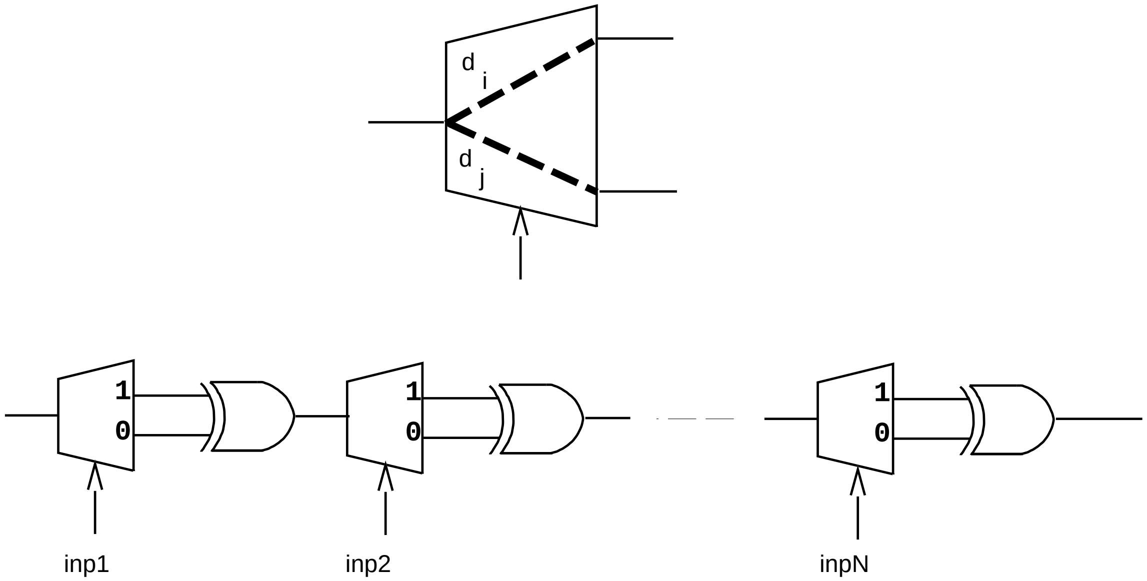


Figure 3-18: The demultiplexer circuit, used to test the feasibility of additive delay modeling of a PUF circuit.

are due to slowly varying parameters such as temperature and supply voltage), we would be able to extract $\frac{1}{2}\log_2\left(\frac{10000}{100}\right) \approx 3.3$ bits per measurement. So with 20 measurements, on 20 different loops, we would get about 2×33 bits, which would allow us to distinguish between 10 billion different chips (taking into account the birthday paradox).

To summarize the experiments in this section, compensated measurements enable reliable identification under appreciable environmental variations.

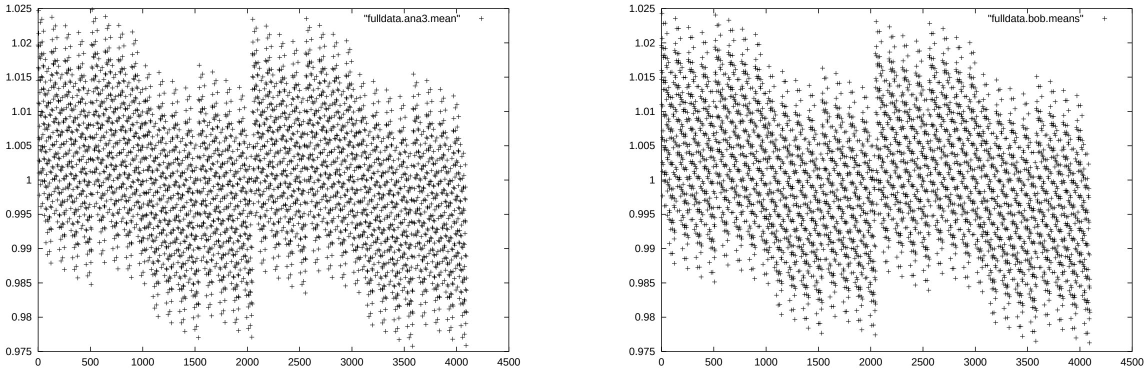
We note that variance in a manufacturing process can be increased quite easily by making small changes in the fabrication steps, e.g., not regulating temperature and pressure as tightly, and increased variance will allow reliable identification under a greater amount of environmental variation. Also, with the advent of deep sub-micron (e.g., 90 nm) devices, there is greater intrinsic fluctuation for minimum width devices due to lithography tolerance and dopant fluctuation [28]. Finally, an IC containing a PUF could be placed in an environment-resistant board to improve reliability.

3.3.2 Additive Delay Model Validity

We ran the same experiments on the demultiplexer circuit shown in Figure 3-18. A circuit with 12 stages was used in our experiments.

The observed measurement error, inter-FPGA variation and dependence on environmental conditions were compatible with the results from Section 3.3.1.

In addition to confirming the results from the previous experiments, the new circuit was able to show us the effect of challenges on the frequency of the self-oscillating loops. Figure 3-19 shows the compensated response of two different FPGAs as a function of the input challenge.



(a) Ana FPGA

(b) Bob FPGA

Figure 3-19: Compensated Delay versus Input Challenges for the Demultiplexer circuit on two different FPGAs: The large scale structure is identical, and is due to differences in routing of paths on a given circuit. The difference between the FPGA appears at a much smaller scale, and can be seen as a difference in texture between the two plots.

There is a clear dependency of the output on the challenge. Moreover, and quite predictably, there is a lot of structure in the challenge-dependence of the response. This structure is common to the two FPGAs and is due to large differences between paths in given stages of the delay circuit. To actually see a difference between the two FPGAs, one must look at the small scale differences between the two plots (we are looking for 1% variations on a plot that covers 50% variations). These differences are present, and appear most clearly as a difference in texture between the plots for the two chips.

The reason why such a simple circuit was chosen for this experiment is that we wanted to quantify how well an adversary could simulate the circuit by choosing an additive delay model. Indeed, suppose that the adversary wanted to create a model for the demultiplexer circuit of Figure 3-18. He reasons that the delay of the circuit under each challenge is the delay of the actuated path for that challenge. He can assume an additive delay model, where the delay of a path is the sum of the delays of the components and wires on that path. By measuring the delay of a set of paths that cover all the components and wires in the circuit, he can set up a linear system of equations that relate the unknown device and wire delays to known path delays. He can then solve for the device and wire delays, thereby obtaining a model of the circuit, which he can then simulate to guess at the response for an arbitrary challenge. The question then is: "How accurate is the model created by the adversary?" If the model is inaccurate, then the adversary can try to augment it by adding non-additive delay behavior or additional variables, and continue. The effort involved in non-additive model building is considerably higher, but difficult to quantify precisely. Here, we will restrict ourselves to quantifying the complexity/error tradeoff of additive model building.

To quantify the accuracy of an additive model that the adversary can build, we measured the delays of all 2^n paths in a $n = 12$-stage demultiplexer circuit. Each of these paths corresponds to a different challenge. For a pair of paths P_1 and P_2 whose challenges differ in exactly one bit, the paths coincide in all stages but one. The adversary may assume an additive delay model which implies that the relationship between the path delays is

$$P_1 \; - \; P_2 \; = \; d_i \; - \; d_j \; .$$

The d_i and d_j pairs are marked on Figure 3-18.

Using all 2^n measured delays, we determined a mean and standard deviation for each of the $d_i - d_j$ quantities. This standard deviation is characteristic of the inaccuracy of the additive model, and we shall call it σ_{calc}. In our experiments, σ_{calc} was between 5 *ppm* and 30 *ppm*, which is roughly the same as the environmental variations that we have to deal with. Thus, the additive model might be a valid way of breaking simple circuits such as the one in Figure 3-18.

Nevertheless, even if the additive delay model gives results that are within the tolerances that the adversary has to meet, he may not be able to use it to efficiently simulate the circuit. Indeed, when he uses the additive delay model, the adversary is essentially starting from a challenge he knows a response to, and performing a certain number of modification steps to the corresponding delay to account for differences between the known challenge and the one he is trying to calculate the response for. The modeling error, σ_{calc}, is present for each one of the additions that the adversary performs. It is likely that the error that is committed when the model is applied multiple times will be greater than the best-case error that we have evaluated.

For example, if we assume that the errors that the adversary commits at each step of his computation are Gaussian and independently distributed between steps, then for a k step computation, the adversary in fact commits an error of $\sqrt{k}\sigma_{calc}$. The number of measurements that the adversary would have to make to be able to predict the response to a randomly selected response in fewer than k steps is exponential in $\frac{n}{k}$, so for big enough n, the additive delay model attack will not be sufficient even for simple circuits.

3.4 Possibly Secure Delay Circuit

We now present a circuit that we believe to be breakable in the additive delay model, but that might provide sufficient security on a real chip where the additive delay model is not quite correct.

Tara Sainath and Ajay Sudan have used this circuit to implement the key card application that was described in Section 2.2. In a few seconds, it is able to identify in a secure way any one of the 24 FPGAs we have in the lab. Most of the identification time is spent loading the circuit into the FPGA, so in a hard coded implementation we can expect much faster operation.

3.4.1 Circuit Details

Because we do not have full control over the circuits that are implemented in an FPGA, a few compromises have to be made relative to the theoretical design that required a high

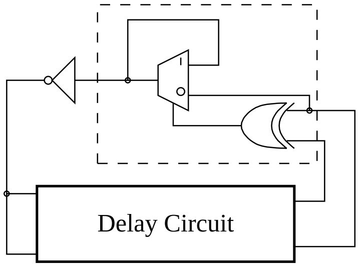


Figure 3-20: A self-oscillating circuit is built around the delay circuit. Measuring the frequency of the self-oscillating loop is equivalent to measuring the delay of a path through the delay circuit.

level of symmetry.

First, the unpredictability of the circuit described in Section 3.2.5 relies on having a circuit with a high level of symmetry between paths. The general purpose routing infrastructure of an FPGA makes it difficult to produce precisely matched paths. Therefore the FPGA circuits that we worked with do not have the degree of symmetry that would be required for a PUF to be secure. However, since the asymmetry is the same across all components, it does not make any change to the difficulty in identifying components, which is what we will be discussing in this section.

Figure 3-20 shows how a self-oscillating loop is built around the simple delay circuit from Figure 3-8. Since both rising and falling transitions are going through the delay circuit, the *and* gate that was used to combine the two paths of the delay circuit in Figure 3-11 has been replaced by a more complicated circuit that switches when the slowest transition, be it rising or falling, reaches it. The circuit is essentially a flip-flop that changes state when both outputs from the delay circuit are at the same level.

The dotted box indicates a delicate part of the circuit that cannot be implemented exactly as shown without running the risk of producing glitching in the output. In the FPGA it is implemented by a lookup table. In an implementation with simple logic, it should be implemented in normal disjunctive form. The representation that was used here was simply chosen for ease of understanding.

3.4.2 Robustness to Environmental Variation

We shall now look at the impact of environmental perturbations that are large enough to mask out the small manufacturing variations that we are trying to measure. These perturbations must be taken into account if chips are to be identified. We shall look at the perturbations that were observed on this particular PUF implementation.

Temperature and Voltage Compensation

Parameters such as temperature or supply voltage cause variations in delay that are orders of magnitude greater than the manufacturing variations that we are trying to observe. For a 30 degree Celsius change in temperature, the delays vary on the order of 5%. This is to be compared with inter-chip variations that are well below 1% on this size of circuit.

Two different self-oscillating loops were placed on the FPGA. We ran both self-oscillating loops to get two frequencies, and took a ratio of the two frequencies as the PUF's response. Once compensation was applied, the variation with temperature was of the same order of magnitude as the measurement error.

For very large temperature changes (at least greater than 30 degrees Celsius), we can no longer expect to reliably recognize a PUF. The answer to this problem is to characterize the PUF once when it is hot and once when it is cold (more steps are possible for large temperature ranges). During use, one of these two cases will apply, so the PUF will be correctly recognized.

Up to now, we have assumed that temperature is uniform across the integrated circuit. If that is not the case, then temperature compensation is likely not to work well. With the circuit presented here, we were careful to keep the paths heated in a uniform way by carefully selecting the challenges that were applied. Without this precaution, we have observed non-uniform heating which can cause unreliable measurement results. Therefore, we recommend heating circuits in a uniform way during use. Non-uniform heating is likely to be a major problem for CPUFs for which the control logic produces a lot of heat, poorly distributed across the chip.

Interference With Other Sub-Systems

Once again, we looked at the interaction between a self-oscillating loop, and other circuitry on the integrated circuit. Experiments in which we measure the frequency of a loop oscillating alone, or at the same time as the other loop, show that the interference is very small as was demonstrated in section 3.3.1.

There is however one case in which interference is non-negligible. It is the case when the interference is at almost the same frequency as the self-oscillating loop. In that case, the loop's frequency tends to lock on the perturbating frequency. This situation sometimes occurs when we are running two loops simultaneously to provide temperature compensation. If the two frequencies happen to be very close, then the two loops interfere, and the two frequencies become identical. When this happens the PUF's response is exactly unity. The consequences of this interference are more or less severe depending on the exact PUF circuitry that is being considered. The compensated response distribution in Figure 3-21 clearly shows the phenomenon.

In the circuits that we tested, if frequencies were close enough to cause locking on one FPGA for a given challenge, then they were generally close enough to cause locking on most FPGAs. This is due to the asymmetry of the delay circuit that we implemented. The result is that when the response is unity, it is useless for identification purposes, as it will be unity across most FPGAs. Therefore, we should not use challenges for which the response is unity.

For highly symmetrical circuits, it is possible that two identical loops would always have frequencies that are close enough to cause locking. This clearly forces us to run the loops separately. Alternatively, we could choose to use non-identical circuits, so that the frequencies will never be close enough to cause locking. The drawback of this approach is

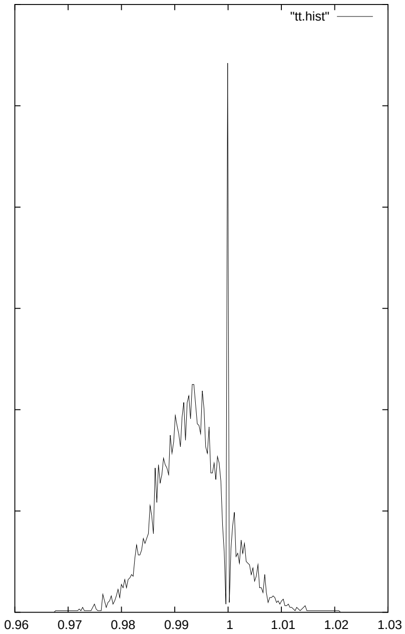


Figure 3-21: Distribution of responses to randomly selected challenges. Each response is the ratio of the frequencies of two simultaneously-running loops. As can be seen, when the loop frequencies are too close, the loops lock and the response is unity.

that we have found temperature compensation to work better when the circuits that are being used for compensation are more alike.

Overall, it would seem that it is safer to measure loop frequencies one at a time. Fortunately, interference with other parts of the integrated circuit appears to be negligible so no other precautions appear to be necessary.

Aging

Through prolonged use, the delays of an integrated circuit are known to shift. We have not yet studied the effect that aging might have on a PUF. In particular, if the changes due to aging are big enough, we might not be able to recognize a PUF after it has undergone much use. Studying these aging effects is an important aspect that must be covered by future work.

3.4.3 Identification Abilities

To test our ability to distinguish between FPGAs, we generated a number of profiles for many different FPGAs in different conditions. A profile is made up of 128 CRPs. All the profiles were established using the same challenges.

Two profiles can be compared in the following way: For each challenge look at the difference between the responses. You can then look at the distribution of these differences. If most of them are near zero, then the profiles are close. If they are far from zero then the

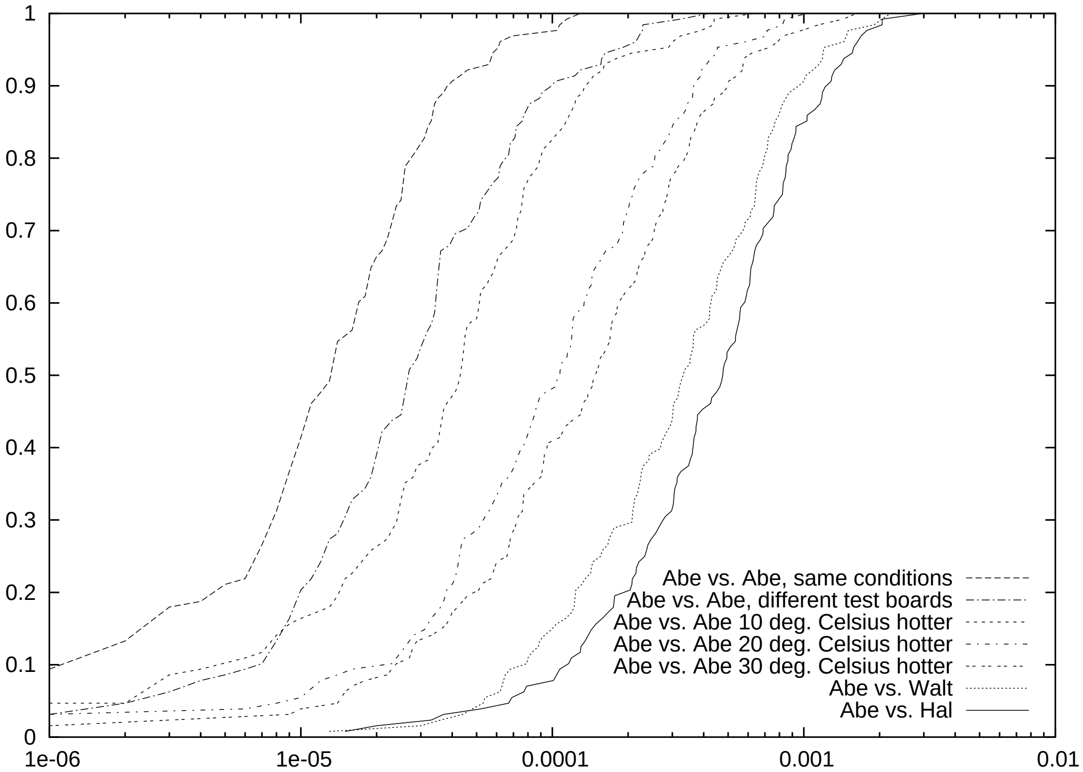


Figure 3-22: Comparing the FPGA called Abe at room temperature with itself in various conditions, or with other FPGAs. The vertical axis indicates the probability that for a given challenge, the difference in response will be lower than the difference in response that is indicated on the horizontal axis. These plots illustrate the typical behavior we encountered in our experiments with many FPGAs.

profiles are distant. During our experiments, the distribution of differences was typically Gaussian, which allows us to characterize the difference between two profiles by a standard deviation.

Figure 3-22 shows the differences between the profile for an FPGA called Abe on Blaise's test board at room temperature, and a number of other profiles (σ is the standard deviation):

- Another profile of Abe on Blaise's test board at room temperature ($\sigma \approx 1 \cdot 10^{-5}$). (This reflects environmental variations with time at a card reader.)

- A profile of Abe on Tara's test board at room temperature ($\sigma \approx 2.5 \cdot 10^{-5}$). (This reflects power supply variations across card readers.)

- Profiles of Abe on Blaise's test board at 10, 20 and 30 degrees Celsius above room temperature ($\sigma \approx 5 \cdot 10^{-5}$ to $1.5 \cdot 10^{-4}$).

- Profiles of FPGAs Hal and Walt on Blaise's test board at room temperature ($\sigma \approx 4 \cdot 10^{-4}$).

The above standard deviations were typical across different FPGAs and comparisons of different pairs of FPGAs.

Clearly, it is possible to tell FPGAs apart. Though our ability to tell them apart depends on how much environmental variation we need to be robust to. Even with 30 degree Celsius variations, each challenge is capable of providing 0.7 bits of information about the identity of the FPGA. This goes up to 1.5 bits if only 10 degree Celsius variations are allowed.

If we want to distinguish between 1 billion different components we need a sufficient number of bits to identify $10^{18} \approx 2^{60}$ components (this is because of the birthday paradox). Getting those 60 bits of information requires from 40 to 90 challenges depending on the temperature variations that we are willing to tolerate.

The numbers that are given here are very dependent on the PUF circuit that is considered. In particular, for the circuits in Section 3.3, we had a better signal to noise ratio than for the current circuit. We believe that by paying more attention to how our circuit is laid out, we will be able to build PUFs for which more bits can be extracted from each challenge.

Chapter 4

Strengthening a CPUF

The weak PUFs that we have been considering so far extract identification information from a physical system, in a way that is somewhat hard for an adversary to predict. They can also be coupled with some control logic, but so far nothing has been said about what the control logic should be. We shall now see that that logic can be combined with a weak PUF to make a much stronger and more reliable PUF.

In each case, we have a PUF f that we are trying to improve in some way. Control allows us to improve f by constructing a new PUF g, that is based on f. The control only allows f to be evaluated as part of an evaluation of g, and only allows the result of the evaluation of f to be used in the evaluation of g.

The block diagram in Figure 4-1 shows most of the improvements that are discussed in this section. For the improvements to be robust to physical attack, the control logic must be intertwined with the PUF so that an adversary can't bypass the logic through physical probing or tampering. In particular he must be prevented from reading the PUF's response directly before it goes through the output random function, and from bypassing the input random function by driving the PUF's challenge directly.

4.1 Preventing Chosen Challenge Attacks

Unless one ventures into quantum effects,[1] the number of physical parameters that define a PUF is proportional to the size of the system that defines it. Therefore, in principle, if an attacker is able to determine a number of primitive parameters that is proportional to the size of the physical system, he can use them to simulate the system and thus clone the PUF.

To try to determine primitive parameters, the attacker can get a number of CRPs, and use them to build a system of equations that he can try to solve. By definition, for a PUF, these equations should be impossible to solve in reasonable time. However, there can be physical systems for which most CRPs lead to unsolvable equations, while a small subset of CRPs give equations that are able to break the PUF (which consequently is not really a PUF). Such a system is not secure because an adversary can use the CRPs that lead to simple

[1] For quantum effects to make a difference, a long term coherence of the relevant quantum state would be necessary for it to belong to the identity of a device. Long term coherence is something that is certainly out of reach in the near future.

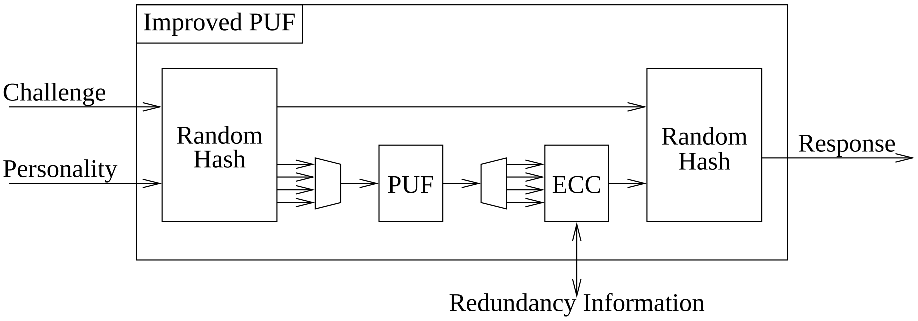


Figure 4-1: This diagram shows how control can be used to improve a PUF. Random functions are used at the input and output of the PUF, an Error Correcting Code is used to make the PUF reliable, vectorization is used to extract sufficient identification information for each challenge, and a personality selector allows the owner of the PUF to maintain his privacy.

equations to get a solvable system of equations, calculate the primitive parameters, and clone the PUF by building a simulator. Such a challenge would be called a chosen-challenge attack.

With control, building a secure system out of one of a PUF with this weakness is possible. For example, the control layer can simply refuse to give responses to challenges that lead to simple equations. Unfortunately, doing so requires us to know all the strategies that the attacker might use to get a simple set of equations from a chosen set of CRPs.

We can do better if we pre-compose the broken PUF with a random function:[2] instead of using f directly, we use

$$g(x) = f(h(x)),$$

where h is a random function. With this method, it is impossible for the adversary to choose the challenges $h(x)$ that are being presented to the underlying PUF, so even if he finds a set of challenges that would break the PUF, he is unable to present those challenges. In this manner, a PUF can be protected from any chosen-challenge attack, without any knowledge of the details of any particular attack.

A variant to using a random function is to use a distance d encoder. Such an encoder implements a mapping such that images of different elements always differ on at least d bits, which means that at least d bits of the input to the PUF wouldn't be up to the attacker. There are many implementations of distance d encoders, while unkeyed pseudo-random functions are only an approximation of a random function, so distance d encoders are a theoretically more satisfying approach.

Moreover, for some encoder implementations, the output values of the encoder can be correlated. This correlation might be sufficient to break some statistical modeling attacks that Marten van Dijk has been considering.[3] A combination of a random function followed

[2] In an actual implementation, all the random functions that we talk about would be implemented with pseudo-random functions such as SHA1 [20] or MD5 [22].

[3] An example of a statistical modeling attack on delay-based PUFs is to query a random set of CRPs, and then compute expected values of the delays with constraints on a few of the challenge bits. Comparing

by an distance d encoder is also a possibility, which would combine the advantages of both methods.

4.2 Vectorizing

As we have seen in sections 3.3 and 3.4 only a few bits of identification information are actually extracted from each call to the weak PUF. That means that an adversary with access to another PUF of the same type only has a few bits to guess to get the output of the PUF. In practice, we would like the adversary to have to guess at least 64 to 128 bits before he can get the actual output. Therefore, to make a strong PUF, each query to the strong PUF must query the weak PUF multiple times so that sufficiently many identification bits can be extracted from the PUF to prevent brute force attacks. We call this technique *vectorization*.

Implementation is relatively easy, we simply choose the input random function so that it provides a very wide output. This output is split into many different challenges for the weak PUF. The challenges are fed through the weak PUF one at a time, and the responses are concatenated into a single response that contains enough identification information.

4.3 Post-Composition with a Random Function

The output of a PUF is, by definition, supposed to resemble the output of a random function. However, the output of the weak PUF we have considered is likely to contain many patterns, as was illustrated by Figure 3-19. Moreover, as we discussed in Section 4.1, CRPs reveal information about the physical system that generates the PUF, and so can be used in a modeling attack to get systems of equations over the PUF's underlying physical parameters.

Both of these problems can be eliminated by doing another simple transformation on the PUF. If f is the PUF that we are trying to improve, and h is a random function, then

$$g(x) = h(x, f(x))$$

is a stronger PUF. Indeed, g is a random function, and as long as $f(x)$ contains enough identifying bits to prevent brute force attacks (the adversary computes $g(x)$ assuming each possible value for $f(x)$), the fact that g is public doesn't help the adversary. Moreover, knowing $g(x)$, the adversary can check if a value for $f(x)$ that he has hypothesized is correct, but cannot get any information to help him find $f(x)$. Thus, the physical meaning of the response that model-building attackers want to use has been stripped away in $g(x)$.

Post-composing the PUF with a random function is a very important step because it makes the system provably resistant to non-physical attacks, as long as enough information is extracted from the physical system before running it through the output random function to prevent brute force attacks. Thanks to vectorization, that minimum amount of information can easily be reached.

expected values for different constraints can provide a lot of information about individual component delays.

4.4 Giving a PUF Multiple Personalities

A possible concern with the use of PUFs is in the area of privacy. Indeed, past experience shows that users feel uncomfortable with processors that have unique identifiers, because they are afraid that the identifier will be used to track them. Users could have the same type of concern with the use of PUFs, given that PUFs are a form of unique identifier.

This problem can be solved by providing a PUF with multiple personalities. The owner of the PUF has a parameter that she can control that allows her to show different facets of her PUF to different applications. To do this, we hash the challenge with a user-selected personality number, and use that hash as the input to the rest of the PUF.

In this way, the owner effectively has many different PUFs at her disposal, so third parties to which she has shown different personalities cannot determine if they interacted with the same PUF.

We go into the details of protocols that use multiple personalities in Section 5.1.2.

4.5 Error Correction

In many cases, the PUF is being calculated using an analog physical system. It is inevitable that slight variations from one run to the next will cause slight changes in the digitized output of the PUF. This means that the chip only produces an approximation of the response that is expected of it. In most applications, the chip and the challenger cannot directly compare the real response with the desired response as this would require sending one of the responses in the clear, thus compromising the shared secret. Therefore, something must be done to make the PUF's output identical each time a challenge is reused.

A suitably selected error correcting code is the typical answer to this type of problem. When a challenge-response pair is created, some redundant information is also produced that should allow slight variations in the measured parameters to be corrected for. On subsequent uses of the challenge-response pair, the redundant information is provided to the PUF along with the challenge. It is used to correct the response from the physical system. It is of course critical that the redundancy information not give away all the bits of the response. In addition to the redundancy information, the attacker is assumed to know the distribution of measurements across PUFs, as this information can be derived from the PUF's specification.

Naturally, the error correction must take place directly on the measured physical parameters. In particular, if the PUF is post-composed with a random function, the correction must take place first. If multiple measurements are being combined into one response, the error correction should operate on all the measurements. Unfortunately, since the redundancy information originates before the output random function, it will reveal information about the underlying physical system. We have not yet found ways of avoiding that problem.

Selecting the best error correcting code for a problem is not an easy task. We shall present a fairly simple code to prove that it is possible. This code is designed with the circuit from Section 3.4 in mind.

4.5.1 Discretizing

To simplify our task, we will attempt to extract a single bit b of information from each compensated measurement, which we will then do error correction on. We will essentially do this extraction by quantizing the measured value with a step size of δ, and taking the resulting value modulo 2.

Let d be the compensated measurement that is computed when the redundancy information is created, and m the compensated measurement that is computed when the redundancy information is used. From what has been said, b would be given by $b = \lfloor \frac{m}{\delta} \rfloor \bmod 2$. Unfortunately, when $\frac{d}{\delta}$ is nearly integer, a little noise is enough to make the bit derived form m different from the bit derived from d, thus accounting for a large proportion of the overall bit error rate.

To avoid this problem, we slightly change the definition of b to $b = \lfloor \frac{m-\epsilon}{\delta} \rfloor \bmod 2$ where $\epsilon = \delta - \lfloor \delta \rfloor - \frac{1}{2}$. That way, d is right in the middle of the quantization interval, and our chances of m being quantized the same way as d are maximized. However, we now have to send ϵ as part of the redundancy information, so we are giving all the low order bits of d to a potential adversary. Unfortunately, there doesn't yet seem to be a good way around this problem.

We shall assume here that the bits of ϵ don't give an adversary any information about the bit b that is extracted from d. The assumption is in fact wrong when δ grows to be near the standard deviation of d across chips. The right choice for δ will be discussed in Section 4.5.3.

4.5.2 Correcting

Now that we have converted the compensated measurements into bits, we will try to correct the bits to get rid of all errors. We will do that using a product of a modified Hamming code [13] and a Parity check.

Modified Hamming Codes

We use a variant of Hamming codes. The variation we use is possible because, in our application, there are no errors in the redundancy bits. To compute the modified Hamming code of a $2^k - 1$ bit message represented by a column vector over the order two finite field, we multiply it by a k row matrix whose i^{th} column is the binary representation of i. For example, the redundancy information for `1011001` is computed by:

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

The redundancy information for `1011001` is therefore `001`.

This code can correct a single error on non-redundancy bits. To correct an error, compute the redundancy information for the erroneous message, and exclusive-or it with the redundancy information for the correct message. The result is the binary encoding of the offset of the erroneous bit in the message, unless it is zero, in which case there is no error. For example,

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

and $\texttt{010} \oplus \texttt{001} = \texttt{011}$. It is indeed the third bit that had been changed. Of course, if more than two errors are present, the modified Hamming code fails.

By adding a parity bit, it is possible to detect but not correct a second error. The second error is apparent because when two bits are erroneous, the parity bit is correct, but the modified Hamming code indicates an error.

The modified Hamming code can be applied to messages whose length cannot be expressed as $2^k - 1$ simply by padding the message with zeroes.

The Product Code

The modified Hamming code does not provide sufficient correction for our purposes. We will therefore augment it by creating a product code.

We arrange $w \cdot h$ bits into a w-column, h-row array. We use a modified Hamming code and a parity bit on each row, and a parity bit on each column.

As long as there is only one error per row, the modified Hamming codes are able to correct all the errors. However, if one row contains two errors, the Hamming code fails. In that case, the parity bit on the row tells us that the row contains two errors. If only one row contains two errors, the parity bits on the columns allow us to determine which bits of the faulty row are incorrect. If more than one row contains two errors, or one row contains more than two errors, the code fails.[4]

An Optimization

The product code can still be slightly optimized. Indeed, the row parity bits are redundant most of the time because we can directly calculate them from a corrected row of bits. The only case where we cannot calculate them all, but we can still correct all the errors, is when one row contains two errors, and the other rows contain at most one error. In that case, if we calculate the row-parities from the row data, exactly one of the parities will be wrong. That means that instead of storing the parities, we can use a modified Hamming code on

[4] There are in fact a number of other rare cases where correction can still take place. We shall disregard them in our calculations.

the row-parities, and only store the redundancy information on what the row-parities should be. In this way we save a few extra bits.

Constraints

Not any choice of w and h will suit our purposes. We would like to present the output hash with at least B identification bits that the adversary doesn't have. A current typical value of B that avoids brute force attacks is around 80. Moreover, we would like the probability of successfully correcting all the errors, and getting the correct response to be high.

It will turn out that with the current error correction method, and the experimental data we have, the constraint on the number of identification bits cannot be met without lowering the probability of getting the correct response below about 50%. This implies that we will have to adapt CPUF protocols so that they try a few different challenges until the PUF gives the right response to one of them. Trying the same challenge multiple times isn't good enough because the error is predominantly due to slowly changing environmental parameters, rather than noise that changes each time a measurement is made. The quantity we will try to minimize in choosing w and h is B_{exp}, the expected number of measurements to perform on the PUF.

To compute the number of identification bits, we assume an error rate p for the adversary. That error rate tells us the adversary's maximum channel capacity $C = 1 + p \cdot \log_2(p) + (1 - p) \cdot \log_2(1 - p)$. We consider that the adversary has $B_a = C \cdot w \cdot h + R$ bits of information, where $R = w + h \cdot \lfloor \log_2(w) + 1 \rfloor + \lfloor \log_2(h) + 1 \rfloor$ is the number of redundancy bits. The number of identification bits we have extracted from the PUF is therefore the difference between the number of bits in the block, and the number of bits the adversary has: $w \cdot h - B_a$. In general, many blocks of w by h bits must be sent before B bits of identification information are available. We will call B_{tot} the number of bits that are needed to get B information bits.

Computing the probability of correctly correcting all the bits that are needed to gather B information bits, knowing the error rate q for the PUF measurements, is a relatively simple application of Bernoulli distributions, and is left to the reader. Essentially, the reader must compute the probability of correcting a given row and the probability of detecting two errors in a given row. With these probabilities, he can compute the probability of detecting two errors in more than one row and the probability of having more than two errors in any row. These easily give him a lower bound on the probability of correcting a whole block. Knowing how many blocks must be read, one can deduce the probability P_{succ} of getting all the blocks right. From there the expected number of physical measurements to perform can be deduced.

4.5.3 Orders of Magnitude

The data from Figure 3-22 can be used to find values of p and q, given δ. The value of $\frac{\delta}{2}$ corresponds to a vertical line on the graph. For values above about 60%, p and q can be read directly off that line of the graph. For p one should take the value of the highest plot that corresponds to two different FPGAs. For q one should take the value of the lowest plot that corresponds to the same FPGAs, in environmental conditions in which we want to be

Case	$\delta/2$ (ppm)	p	q	h	w	P_{succ}	B_{tot}	B_{exp}
1	≈ 250	55 %	70 %	10	3	$4.7 \cdot 10^{-29}$ %	870	$1.9 \cdot 10^{33}$
2	≈ 500	68 %	90 %	30	3	20 %	540	2681
3	≈ 1500	95 %	99 %	31	30	58 %	930	1617

Figure 4-2: Some parameters for the ECC. Note that in case 1, the value of p is an approximation as the value is too low to be read directly off the graph. In case 3, the value of p is too high for the assumption that the low order bits of the measurement reveal nothing about the bit we extract to be true.

able to recognize it. Some examples are given in Figure 4-2, along with the optimum error correction solution using our code, for those parameters.

The optimum error correction solution is computed by a simple C program that calculates the expected number of physical measurements as a function of w and h. It considers that a whole number of w by h blocks must be used. The program involves a few small approximations, so the numbers presented here are slightly conservative.

Figure 4-2 reveals that it is easier to find a good tradeoff when there are few measurement errors, so δ should be chosen accordingly. Cases 2 and 3 show that as long as the measurement errors are limited, adequate solutions can be found for a wide range of values of δ. Of course, if δ is too large, both p and q are so close to one that error correction is impossible once again.

Assuming a 100 MHz clock, and 2 × 10000 cycles per measurement, on the order of 3 CPUF evaluations can be carried out per second. Clearly there is a need for further improvement.

4.5.4 Optimizations

The main optimization that we can try to make to our error correction strategy, is to extract two or three bits from each compensated measurement by reducing modulo four or eight. Each bit from a measurement corresponds to its own value of δ, and therefore, to its own values of p and q. It is therefore desirable to correct the three levels of bits independently of each other. Each one will have its own settings for w and h, and a global optimization of block sizes should be done. By extracting more information in this way, it should be possible to get away with fewer measurements.

When using multiple bits per measurement, errors will often be correlated. In particular, if a high order bit is found to be wrong, we can expect the lower order bits to be random. Therefore we can consider them as erasures, and try to take the erasure information into account to correct more errors on the low order bits.

4.6 Multiple Rounds

To add even more complexity to the attacker's problem, it would be possible to use the PUF circuit multiple times to produce one response. The corrected response from one round would be fed back into the PUF circuit as a challenge. After a few rounds have been done, all their outputs could get merged together along with the initial challenge, the personality and the chip's identifier and passed through a random hash function to produce the global response.

Chapter 5

CRP Infrastructures

In this chapter, we describe an infrastructure that allows arbitrary remote users to get CRPs for a CPUF equipped device. This infrastructure addresses the PUF equivalent of the key distribution problem. First we will present the architecture, then the corresponding protocols, and finally a few applications.

5.1 Architecture

5.1.1 Main Objective

Figure 5-1 illustrates the basic model we will be working in.

- The user is in the possession a list of CRPs for a CPUF device that she knows to be conforming to its specification. Only the user and the device know the responses to the CRPs on the list. However, we assume that the challenges are public.
- The user and the CPUF device are connected to one another by an untrusted communication channel that is neither private nor authenticated.
- The user wants to share a secret with the CPUF device to engage in some cryptographic protocol with it.
- The adversary would like to know the shared secret so that he can abuse the cryptographic protocol.

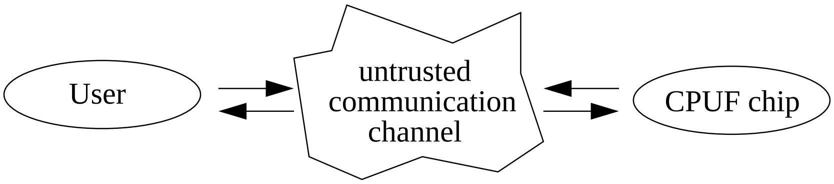


Figure 5-1: Model for Applications

The next section will give an overview of how the user gets her CRPs. To get her CRPs, she will have to share a secret with the PUF. Securing communication between the user and the PUF is easy once they have a shared secret. The last problem, establishing a shared secret, seems to be the central problem that has to be solved.

It turns out that without control, it is impossible to establish a shared secret with the PUF. In fact, having a list of CRPs where the challenges are public but the responses are only known to the user is also impossible without control. In Section 5.2, we shall see some very general control primitives that restrict access to the PUF just enough to make secret sharing with the PUF possible.

5.1.2 CRP Management Primitives

In our models for CRP management, the user does not have CRPs for the CPUF yet, and would like to establish her own private list of CRPs. The following three new principals will participate in establishing that list:

- *Manufacturer*: the manufacturer is the principal that made the CPUF device. Just after making it, the manufacturer had secure access to the device, through direct physical contact. The manufacturer is trusted by everybody because he has the power to embedded a back door in the device.[1]

- *Owner*: the owner is the principal who has the device in his possession, and who can give access to the device to other principals. The owner is assumed to have a private list of CRPs for the device. By extension, it will sometimes be possible for principals other than the legal owner to participate in protocols as the owner. This would happen if the legal owner gives access to his device to Alice, and Alice were then to give Bob access to the device through her connection to it. Alice would be the owner in the second part of the process.

- *Certifier*: the certifier has its own private list of CRPs for the device, is trusted by the user, and wants to help the user get some CRPs of her own.

  The principals mentioned above are by no means absolute. Except for the manufacturer, these principals can be interchanged many times during the lifetime of a PUF.

We have 5 scenarios that allow us to set up an CRP management infrastructure: bootstrapping, renewal, introduction, private renewal and anonymous introduction.

Bootstrapping

Figure 5-2 illustrates bootstrapping. When a CPUF has just been produced, the manufacturer generates a set of CRPs for it. The manufacturer knows that these CRPs refer to a correct PUF device, because he has just built (and tested) the device. Since he is in physical contact with the device in a secure location, there is a secure channel between the manufacturer and the device.

[1] Trust in the manufacturer could be reduced by reverse-engineering a random sample of devices to check that they match their specification.

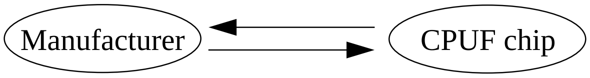


Figure 5-2: Model for Bootstrapping

Renewal

Figure 5-3 illustrates renewal. A user has a private list of CRPs for a CPUF device, and would like to get more CRPs. The user uses a CRP to open a secure channel to the device, and then gets the device to generate a new CRP and return the result. Since the channel is secure, only the user knows the corresponding response. We will see in Section 5.2 why we can assume that that CRP isn't known to somebody else yet.

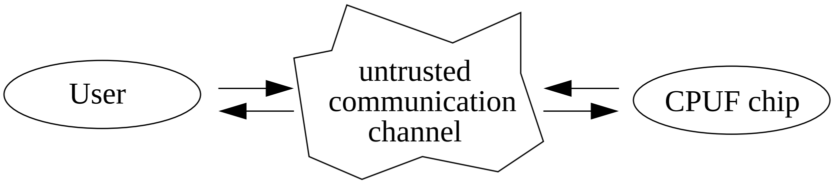


Figure 5-3: Model for Renewal

Introduction

Figure 5-4 illustrates introduction. A user wants to get a CRP for a CPUF device. The certifier has a private list of CRPs for the device. The user trusts the certifier, and has a classical way of opening a secure channel with him.

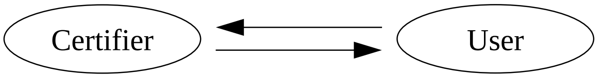


Figure 5-4: Model for Introduction

In introduction, the certifier simply gives a CRP to the user over a secure channel, and removes the CRP from his own list of private CRPs and forgets it.

Private Renewal

The certifier initially knows the CRP that the user is given during introduction. If he doesn't forget that CRP as he should then the user doesn't actually have a private CRP for the device. The result is that the manufacturer can read all the traffic between the user and the device. Private renewal allows the user to distance herself from the certifier.

Figure 5-3 also applies for private renewal. The user is assumed to already have a CRP but she isn't the only person to have that CRP. She would like to have a CRP that she shares with nobody else.

If the certifier is willing to do an active attack, then there is nothing the user can do. That is acceptable, as there must be some trust between the user and the certifier if the user is to believe that the CRP she was given actually corresponds to a real device. Private renewal protects the user from passive attacks from the certifier.

Essentially, the user's CRP is used to open a secure channel with the device (that the certifier can eavesdrop on). The user sends the device her public key. The device then generates a new CRP and sends it back to the user encrypted with her public key. At that point the user has a CRP of her very own. We will see in Section 5.2 why the certifier can't simply ask the device for the response to the user's new challenge.

Anonymous Introduction

Figure 5-5 illustrates anonymous introduction. The user wants to get a CRP of her own for a CPUF device. A certifier, which the user trusts to give her a valid CRP, has a list of CRPs for a whole collection of devices, including the one the user wants a CRP for. The owner of the device also wants the user to get a CRP for his device from the certifier, but he doesn't want anybody (including the certifier) but the user to know who is getting the CRP. The user doesn't trust the owner to give her a valid CRP.

The aim is to keep secret, for privacy reasons, all information about who the owner is letting his CPUF device interact with. The difficulty is that the user wants the certifier to vouch for the CRPs that he is getting. This scenario will seem clearer when it is studied in detail in Section 5.2. Anonymous Introduction is only possible if the certifier has a huge load of certification to do, thus preventing traffic analysis attacks.

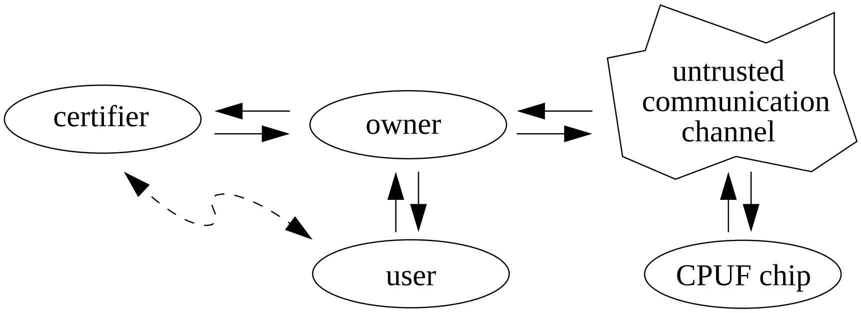


Figure 5-5: Model for Anonymous Introduction

5.1.3 Putting it all Together

Just to clarify how these scenarios fit together, we shall look at a simple example.

One day, a SPUF equipped device called Robo is made at a plant owned by Intal. As the device comes off the production line, *bootstrapping* is used to collect a list of CRPs from it. Later that day, Intal participates in *introduction* protocols with a number of certification agencies, including Verasign, who add Robo to their databases.

A few weeks later, Alice buys Robo, and wants to make sure that it is a genuine product. She contacts Verasign and provides Robo's serial number. Verasign realizes that it has never done *private renewal* with Robo. So it does *private renewal* to get a CRP for Robo that it shares with nobody, and finally *renewal* to increase the list of CRPs that it has. Finally, it provides one of these CRPs to Alice via *introduction*. Alice can use her CRP to check that Robo hasn't been tampered with since it was manufactured. She need not do *private renewal* as she is in physical contact with Robo and therefore has a secure link to it.

Once the novelty of Robo has worn off, Alice decides that she would like to rent access to it to remote users. Bob is interested, he does *introduction* with Verasign to get a CRP for Robo, followed by *private renewal* so that he has a CRP of his very own. He then uses *renewal* to generate more CRPs. Finally he uses some of those CRPs to establish a shared secret with Robo, and gets to work.

Bob soon realizes that he has rented too much of Robo's time, so he would like to sub-rent part of his quota. Charlie is interested in sub-renting, but unfortunately, Bob has agreed with Alice not to do any sub-renting and is afraid of being caught if he disobeys. Fortunately, he can participate in *anonymous introduction*, with himself as the owner, Charlie as the user, and Verasign as the certifier. Charlie uses *renewal* to generate more CRPs from the CRP that he was given, and starts using Robo.

One day, Bob meets Charlie in person, and realizes that he is in fact Alice. Fortunately, the anonymity protocol did its job, and Alice never even realized that Bob had rented some of Robo's time back to her.

The story ends here, but many other twists could have been added. For example, Bob could have acted as a certifier on a day when the connection to Verasign was down; or Alice could have rented Robo to Bob's friend Dave without Bob and Dave realizing that they were both using the same device.

5.2 Protocols

We will now describe the protocols that are necessary in order to use PUFs. These protocols must be designed to make it impossible to get the response to a chosen challenge. Indeed, if that were possible, then we would be vulnerable to a man-in-the-middle attack that breaks nearly all applications. The strategy that we describe is designed to be deterministic and state-free to make it as widely applicable as possible. Slightly simpler protocols are possible if these constraints are relaxed.

5.2.1 Man-in-the-Middle Attack

Before looking at the protocols, let us have a closer look at the man-in-the-middle attack that we must defend against. The ability to prevent this man-in-the-middle attack is *the fundamental difference* between controlled and uncontrolled PUFs.

The scenario is the following. Alice wants to use a CRP that she has, to engage in a cryptographic protocol with a CPUF (we are assuming that the CRP is the only thing that

Alice and the CPUF have in common). Oscar, the adversary, has access to the PUF, and has a method that allows him to extract from it the response to a challenge of his choosing. He wants to impersonate the CPUF that Alice wants to interact with.

At some point, in her interaction with the CPUF, Alice will have to give the CPUF the challenge for her CRP so that the CPUF can calculate the response that it is to share with her. Oscar can read this challenge because up to this point in the protocol Alice and the CPUF do not share any secret. Oscar can now get the response to Alice's challenge from the CPUF, since he has a method of doing so. Once Oscar has the response, he can impersonate the CPUF because he knows everything Alice knows about the PUF. This is not at all what Alice intended.

We should take note that in the above scenario, there *is* one thing that Oscar has proven to Alice. He has proven that he has access to the CPUF. In some applications, such as the key cards in Section 2.2, proving that someone has access to the CPUF is probably good enough. However, for more powerful examples such as certified execution that we will cover in Section 5.3.2, where we are trying to protect Alice from the very owner of the CPUF, free access to the PUF is no longer a sufficient guarantee.

More subtle forms of the man-in-the-middle attack exist. Suppose that Alice wants to use the CPUF to do what we will refer to in Section 5.3.2 as *certified execution*. Essentially, Alice is sending the CPUF a program to execute. This program executes on the CPUF, and uses the shared secret that the CPUF calculates to interact with Alice in a secure way. Here, Oscar can replace Alice's program by a program of his own choosing, and get his program to execute on the CPUF. Oscar's program then uses the shared secret to produce messages that look like the messages that Alice is expecting, but that are in fact forgeries.

Fortunately, all these attacks can be defeated. We shall now see how access to the PUF can be restricted by control to thwart the man-in-the-middle attack.

5.2.2 Defeating the Man-in-the-Middle Attack

Basic CPUF Access Primitives

In the rest of this section, we will assume that the CPUF is able to execute some form of program in a private (nobody can see what the program is doing) and authentic (nobody can modify what the program is doing) way.[2] In some CPUF implementations where we do not need the ability to execute arbitrary algorithms, the program's actions might in fact be implemented in dedicated hardware or by some other means — the exact implementation details make no difference to the following discussion.

In this paper we will write programs in pseudo-code in which a few basic functions are used:

- `Output(arg1, ...)` is used to send results out of the CPUF. Anything that is sent out of the CPUF is potentially visible to the whole world, except during bootstrapping, where the manufacturer is in physical possession of the CPUF.

[2] In fact the privacy requirement can be substantially reduced. Only the key material that is being manipulated needs to remain hidden for these protocols to work.

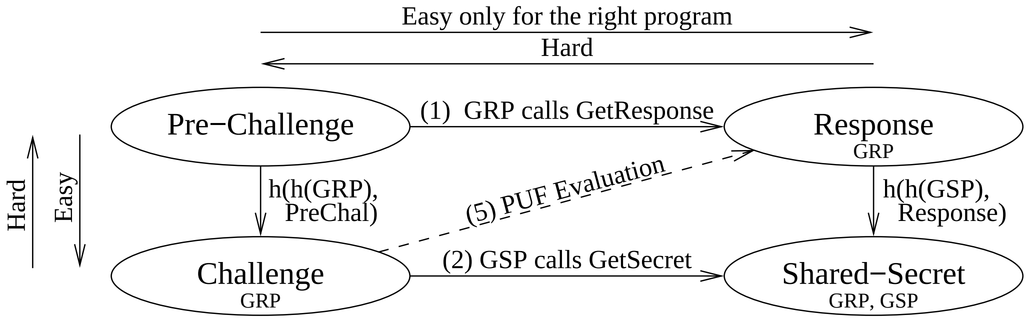


Figure 5-6: This diagram shows the different ways of moving between Pre-Challenges, Challenges, Responses and Shared-Secrets. The dotted arrow indicates what the PUF does, but since the PUF is controlled, nobody can go along the dotted arrow directly. GRP and GSP are the programs that call *GetResponse* and *GetSecret*, respectively. The challenge and the response depend on the GRP that created them, and the shared secret depends on the GSP.

- `EncryptAndMAC(message, key)` is used to encrypt and MAC `message` with `key`.
- `PublicEncrypt(message, key)` is used to encrypt `message` with the public key `key`.
- `MAC(message, key)` MACs `message` with `key`.

The CPUF's control is designed so that the PUF can only be accessed by programs, and only by using two primitive functions: *GetResponse* and *GetSecret*.[3] If f is the PUF, and h is a publicly available random hash function (or in practice some pseudo-random hash function) then the primitives are defined as:

$$GetResponse(PreChallenge) = f\left(h\left(h\left(Program\right), PreChallenge\right)\right)$$

$$GetSecret(Challenge) = h\left(h\left(Program\right), f\left(Challenge\right)\right)$$

In these primitives, $Program$ is the program that is being run. Just before starting the program, the CPUF calculates $h(Program)$, and later uses this value when *GetResponse* and *GetSecret* are invoked. We shall show in the next section that these two primitives are sufficient to implement the CRP management scenarios that were detailed in Section 5.1.2. We shall also see that *GetResponse* is used for CRP generation while *GetSecret* is used by applications that want to produce a shared secret from a CRP. Therefore, in a PUF that uses error correction, *GetResponse* would produce the redundancy information that is needed for error correction, while *GetSecret* would use it (see Section 4.5).

Figure 5-6 summarizes the possible ways of going between pre-challenges, challenges, responses and shared secrets. In this diagram, moving down is easy. You just have to

[3] If a strong random number generator is assumed, then *GetResponse* can actually be replaced by a *GetCRP* function that takes no arguments, and returns a random CRP. The *GetCRP* function would be used everywhere the *GetResponse* function is used. It is not clear what advantage this interface would have, the main difference being that a strong random-number generator is now needed. We have mentioned it mainly for completeness.

calculate a few hashes. Moving up is hard because it would involve inverting hashes that are one-way. 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 if we assume that the PUF can't invert a one-way function. We will not use this fact as the adversary's task wouldn't be easier if it was easy.

Using a CRP to Get a Shared Secret

To show that the man-in-the-middle attack has been defeated, we shall show that a user who has a CRP can use it to establish a shared secret with the PUF (previously, the man-in-the-middle could determine the value of what should have been a shared secret).

The user sends a program like the one below to the CPUF, where `Challenge` is the challenge from the CRP that the user already knows.

```
begin program
  Secret = GetSecret(Challenge);
  /* Program that uses Secret as a shared secret with the user. */
end program
```

Note that $h\left(program\right)$ includes everything that is contained between `begin program` and `end program`. That includes the actual value of `Challenge`. The same code with a different value for `Challenge` would have a different program hash.

The user can determine `Secret` because he knows the response to `Challenge`, and so he can calculate $h\left(h\left(program\right), response\right)$. Now we must show that a man-in-the-middle cannot determine `Secret`.

By looking at the program that is being sent to the CPUF, the adversary can determine the challenge from the CRP that is being used. This is the only starting point he has to try to find the shared secret. Unfortunately for him, the adversary cannot get anything useful from the challenge. Because the challenge is deduced from the pre-challenge via a random function, the adversary cannot get the pre-challenge directly. Getting the Response directly is impossible because the only way to get a response out of the CPUF is starting with a pre-challenge. Therefore, the adversary must get the shared secret directly from the challenge.

However, only a program that hashes to the same value as the user's program can get from the challenge to the secret directly by using *GetSecret* (any other program would get a different secret that can't be used to find out the response or the sought after secret because it is the output of a random function). Since the hash function that we are using is collision resistant, the only program that the attacker can use to get the shared secret is the user's program. If the user program is written in such a way that it does not leak the secret to the adversary, then the man-in-the middle attack fails. Of course, it is perfectly possible that the user's program could leak the shared secret if it is badly written. But this is a problem with any secure program, and is not specific to PUFs. Our goal isn't to prevent a program from giving away its secret but to make it possible for a well written program to produce a shared secret.

5.2.3 Challenge Response Pair Management Protocols

Now we shall see how *GetResponse* and *GetSecret* can be used to implement the key management primitives that were described in Section 5.1.2.[4] It is worth noting that the CPUF need not preserve any state between program executions.

Bootstrapping

The manufacturer makes the CPUF run the following program, where `PreChall` is set to some arbitrary value.

```
begin program
  Response = GetResponse(PreChall);
  Output(Response);
end program
```

The user gets the challenge for his newly created CRP by calculating $h(h(\mathit{program}), \texttt{PreChall})$, the response is the output of the program.

Renewal

The user sends the following program to the CPUF, where `PreChall` is set to some arbitrary value, and `OldChall` is the challenge from the CRP that the user already knows.

```
begin program
  NewResponse = GetResponse(PreChall);
  Output(EncryptAndMAC(NewResponse, GetSecret(OldChall)));
end program
```

Only the user and the CPUF have the initial CRP needed to compute `GetSecret(OldChall)`. It is their shared secret. The user can be sure that only he can get `NewResponse`, because it is encrypted with the shared secret. An adversary can change `OldChall` to a challenge that he knows the response to, but since `OldChall` is part of the program, the newly created CRP would be different from the one that the adversary is trying to hijack (because *GetResponse* combines the pre-challenge with a random hash of the program that is being run). The MAC proves that the `NewResponse` that the user is getting originated from the CPUF. The user gets the challenge for his newly created CRP by calculating $h(h(\mathit{program}), \texttt{PreChall})$.

[4] The implementations that are presented contain the minimum amount of encryption to ensure security. A practical implementation would probably want to include nonces to ensure message freshness, and would encrypt and MAC as much information as possible. In particular, it is not necessary in our model to encrypt the pre-challenges that are used to produce CRPs. Nevertheless hiding the pre-challenge (and therefore the challenge) would make it harder for an adversary to mount an attack in which he manages to forcibly extract the response to a specific challenge from the CPUF.

Introduction

Introduction is particularly easy. The certifier simply sends a CRP to the user over some agreed upon secure channel. In many cases, the certifier will use renewal to generate a new CRP, and then send that to the user. The user will then use private renewal to produce a CRP that the certifier does not know.

Private Renewal

The user sends the following program to the CPUF, where `PreChall` is set to some arbitrary value, `OldChall` is the challenge from the CRP that the user already knows, and `PubKey` is the user's public key.

```
begin program
  NewResponse = GetResponse(PreChall);
  Message = PublicEncrypt(NewResponse, PubKey);
  Output(Message, MAC(Message, GetSecret(OldChall)));
end program
```

The user can be certain that only he can read the `NewResponse`, because it is encrypted with his public key. If the adversary tries to replace `PubKey` by his own public key, he will get the response to a different challenge because `PubKey` is part of the program, and therefore indirectly changes the output of *GetResponse*. The MAC can only be forged by the party that the user is sharing the old CRP with (probably a certifier that the user just performed introduction with). If we assume that that party is not doing an active attack, then we know that the MAC was produced by the CPUF, and therefore, the `NewResponse` is indeed characteristic of the CPUF. The user gets the challenge for his newly created CRP by calculating $h(h(program), \texttt{PreChall})$.

5.2.4 Anonymity Preserving Protocols

In Section 4.4, we showed how a CPUF could be made to take on many different personalities in order to preserve the anonymity of its owner. People don't want their CPUF to give away the fact that the same person is gambling on gambling.com and doing anonymous computation for SETI@home. In this section, we shall add a personality selector to the PUF as in Figure 4-1. We shall call the personality selector `PersonalitySel`. The person who is trying to hide his identity will be called the owner of the CPUF, but as we shall see at the end of Section 5.2.4 the notion is more general than this. We shall assume that all sources of information concerning the identity of the CPUF's owner have been eliminated by other protocol layers, and shall focus on preventing the CPUF from leaking his identity. We shall also assume that there are enough people using anonymized introduction that traffic analysis (correlating the arrival of a message at a node with the departure of a message a little while later simply from timing considerations) is unusable.

Programs must not be given permission to freely write to `PersonalitySel`, or else they could put the CPUF into a known personality and defeat the purpose of having a personality selector. We shall therefore describe how the value of PersonalitySel is controlled. First, two new primitive functions are provided by the CPUF:

- ChangePersonality(Seed) sets the personality to h(PersonalitySel, Seed). Where h is a random hash function.
- RunProg(Program) runs the argument without changing PersonalitySel.

Moreover, when a program is loaded into the CPUF from the outside world, and run (as opposed to being run by RunProg), PersonalitySel is set to zero. We shall call this the default personality.

The pseudo-code uses a few extra primitive functions:

- Decrypt(mesg, key) is used to decrypt mesg that was encrypted with key.
- HashWithProg(x) computes $h(h(program)$, x). This function reads the area where the CPUF is storing the hash of the program.
- Hash(...) is a random hash function.
- Blind(mesg,fact) is used to apply the blinding factor fact to mesg. See Section 5.2.4 for a brief description of blinding.

Choosing the Current Personality

When the CPUF's owner wants to show a personality other than his CPUF's default personality, he intercepts all programs being sent to the CPUF and encapsulates them in a piece of code of his own:

```
  ESeed = /* the personality seed encrypted with Secret */
  EProgram = /* the encapsulated  program encrypted with Secret */

begin program
  Secret = GetSecret(Challenge);
  Seed = Decrypt(Eseed, Secret);
  Program = Decrypt(EProgram,Secret);

  ChangePersonality(Seed);
  RunProg(Program);
end program
```

Note that the line that precedes begin program is a piece of data that accompanies the program but that does not participate in the hash of the program. If EProgram were included in the hash, then we would not be able to encrypt it because the encryption key would depend on the encrypted program. Other values that appear are Seed, an arbitrarily selected seed; and Challenge, the challenge of one of the owner's CRPs.

By encapsulating the program in this way, the owner is able to change the personality that the CPUF is exhibiting when it runs the user's program. There is no primitive to allow the user's program to see the personality that it is using, and the seed that is used with `ChangePersonality` is encrypted so the user has no way of knowing which personality he is using. The user's program is encrypted, so even by monitoring the owner's communication, the user cannot determine if the program that is being sent to the CPUF is his own program.

Anonymous Introduction

The anonymous introduction protocol is much more complicated than the other protocols we have seen so far. We will only sketch out the details of why it works. This protocol uses blinding, a description of which can be found in [24].

The essential idea of blinding is this: Alice wants Bob to sign a message for her, but she does not want Bob to know what he has signed. To do this Alice hides the message by applying what is called a blinding factor. Bob receives the blinded message, signs it and returns the signed blinded message to Alice. Alice can then remove the blinding factor without damaging Bob's signature. The resulting message is signed by Bob, but if Bob signs many messages, he cannot tell which unblinded message he signed on which occasion.[5]

Here is the anonymous introduction protocol:

1. The owner collects a challenge from the certifier, and the user's public key. He produces the following program from Figure 5-7 that is sent to the CPUF.

2. The owner decrypts the output from the CPUF, checks the MAC, and passes `Mesg5` on to the certifier, along with a copy of the program (only the part that participates in the MAC) encrypted with the certifier's public key.

3. The certifier decrypts the program, checks that it is the official anonymous introduction program, then hashes it to calculate `CertSecret`. He can then verify that `Mesg4` is authentic with the MAC. He finally signs `Mesg4`, and sends the result to the owner.

4. The owner unblinds the message, and ends up with a signed version of `Mesg3`. He can check the signature, and the MAC in `Mesg3` to make sure that the certifier isn't communicating his identity to the user. He finally sends the unblinded message to the user. This message is in fact a version of `Mesg3` signed by the certifier.

5. The user checks the signature, and decrypts `Mesg2` with his secret key to get a CRP.

Remarks:

- `UserPubKey` and `CertChallenge` must be encrypted, otherwise it is possible to correlate the message that Alice sends to the CPUF with the certifier's challenge or with the user's public key.

[5] In this protocol, to avoid over-complication, we have assumed that Alice does not need to know Bob's public key in order to sign a message. For real-world protocols such as the one that David Chaum describes in [7] this is not true. Therefore, an actual implementation of our anonymous introduction protocol might have to include the certifier's public key in the program that is sent to the CPUF. In that case, it should be encrypted to prevent correlation of messages going to the CPUF with a specific transaction with the certifier.

```
 /* Various values encrypted with OwnerSecret. */
 ESeed = ...
 EPreChallengeSeed = ...
 EUserPubKey = ...
 ECertChallenge = ...

begin program
 OwnerSecret = GetSecret(OwnerChallenge);
 Seed = Decrypt(ESeed, OwnerSecret);
 PreChallengeSeed = Decrypt(EPreChallengeSeed, OwnerSecret);
 UserPubKey = Decrypt(EUserPubKey, OwnerSecret);
 CertChallenge = Decrypt(ECertChallenge, OwnerSecret);

 CertSecret = GetSecret(CertChallenge);
 PreChallenge = Hash(UserPubKey, PreChallengeSeed);
 NewChallenge = HashWithProg(PreChallenge);
 ChangePersonality(Seed);
 NewResponse = GetResponse(PreChallenge);

 Mesg1 = (NewChallenge, NewResponse);
 Mesg2 = PublicEncrypt(Mesg1, UserPubKey);
 Mesg3 = (Mesg2, MAC(Mesg2, OwnerSecret));
 Mesg4 = Blind(Mesg3, OwnerSecret);
 Mesg5 = (Mesg4, MAC(Mesg4, CertSecret));
 Mesg6 = EncryptAndMAC(Mesg5, OwnerSecret);
 Output(Mesg6);
end program
```

Figure 5-7: The anonymous introduction program.

- `Seed` must be encrypted to prevent the certifier or the user from knowing how to voluntarily get into the personality that the user is being shown.

- `PreChallengeSeed` must be encrypted to prevent the certifier from finding out the newly created challenge when he inspects the program in step 3.

- The encryption between `Mesg5` and `Mesg6` is needed to prevent correlation of the message from the CPUF to the owner and the message from the owner to the certifier.

Interestingly, we are not limited to one layer of encapsulation. A principal who has gained access to a personality of a CPUF through anonymous introduction can introduce other parties to this PUF. In particular, he can send the signed CRP that he received back to the certifier and get the certifier to act as a certifier for his personality when he anonymously introduces the CPUF to other parties.

5.2.5 Protocols in the Open-Once Model

In designing these protocols, we have assumed that the adversary is unable to read or tamper with the digital logic contained in the device. In fact, it very easy to extend the protocols to the slightly less restrictive open-once model from Section 2.3.3.

In the open-once model, the private and authentic execution environment no longer stands, since an invasive adversary can choose to open the device at any time to read and tamper at will. However, the next time the PUF is used, it will return an incorrect value. A user who knows two CRPs can exploit this fact to check all his transactions just after they have been performed:

1. He uses his first CRP to carry out the transaction he initially intended to do.

2. He uses his second CRP in a transaction in which the CPUF device simply proves that it knows the response.

If the second transaction completes successfully, the user knows that the first one took place in a private and authentic execution environment. Indeed, assume that the adversary doesn't initially know the CRPs. Then in order to tamper with the first transaction or eavesdrop on it, he has to open the device while the transaction is taking place. In that case, the second transaction must fail because the PUF is broken, and the adversary doesn't know the second CRP. So by contradiction, the first transaction was carried out in a private and authentic way. The assumption that the adversary doesn't know the initial CRPs can be shown by induction on the sequence of transactions that provided the user with his two CRPs in the first place, assuming that the proper protocols were followed as described in Section 5.2.

5.3 Applications

We believe there are many applications for which CPUFs can be used, and we describe a few here. Other applications can be imagined by studying the literature on secure coprocessors, in particular [29]. We note that the general applications for which this technology can be used include all the applications today in which there is a single symmetric key on the chip.

5.3.1 Smartcard Authentication

The easiest application to implement is authentication. One widespread application is smartcards. Current smartcards have hidden digital keys that can be extracted using various attacks [4]. 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. With a today's cards, 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.

The following basic protocol is an outline of a protocol that a bank could use to authenticate messages from PUF smartcards. This protocol 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.

1. The bank sends the following program to the smartcard, where R is a single use number and Challenge is the bank's challenge:

   ```
   begin program
     Secret = GetSecret(Challenge);
     /* The smartcard somehow generates Message to send to the bank. */
     Output(Message, MAC((Message, R), Secret));
   end program
   ```

2. The bank checks the MAC to verify the authenticity and freshness of the message that it gets back from the PUF.

The number R is useful in the case where the smartcard has state that is preserved between executions. In that case, it is important to ensure the freshness of the message.

If the privacy of the smartcard's message is a requirement, the bank can also encrypt the message with the same key that is used for the MAC.

5.3.2 Certified execution

At present, computation power is a commodity that undergoes massive waste. Most computer users only use a fraction of their computer's processing power, though they use it in a bursty way, which justifies the constant demand for higher performance. A number of organizations, such as SETI@home and distributed.net, are trying to tap that wasted computing power to carry out large computations in a highly distributed way. This style of computation is unreliable as the person requesting the computation has no way of knowing that it was executed without any tampering.

With chip authentication, it would be possible for a certificate to be produced that proves that a specific computation was carried out on a specific chip. The person requesting the computation can then rely on the trustworthiness of the chip manufacturer who can vouch that he produced the chip, instead of relying on the owner of the chip.

There are various ways in which the system could be used. The computation could be done directly on the PUF equipped chip. In that case, the computing power is relatively low, as is often the case for single chip applications. Alternatively, it can be done on a faster, insecure chip that is being monitored in a highly interactive way by supervisory code on the secure chip [29]. The best combination of performance and security can probably be reached by using a PUF equipped processor that uses checked external RAM [10].

To illustrate this application, we present a simple example in which the computation is done directly on the chip. A user, Alice, wants to run a computationally expensive program over the weekend on Bob's single-chip, single-task computer. The single chip on Bob's computer contains a CPUF, and Alice has already established CRPs with the PUF chip.

1. Alice sends the following program to the CPUF, where `Challenge` is the challenge from her CRP:

    ```
    begin program
      Secret = GetSecret(Challenge);
      /* The certified computation is performed, the result is placed in Result. */
      Output(Result, MAC(Result, Secret));
    end program
    ```

2. The Alice checks the MAC to verify the authenticity of the message that it gets back from the PUF.

Unlike the smartcard application, we did not include a single use random number in this particular protocol. This is because we are assuming that we are doing pure computation that cannot become stale (any day we run the same computation, it will give the same result).

In this application, Alice is trusting that the chip in Bob's computer performs the computation correctly. This is easier to ensure if all the resources used to perform the computation (memory, CPU, etc.) are on the CPUF chip, and intertwined with the PUF. We are currently researching and designing more sophisticated architectures in which the CPUF chip can securely utilize off-chip resources using some ideas from [16] and a memory authentication scheme that can be implemented in a hardware processor [11].

There is also the possibility of a central CPUF chip using the capabilities of other networked CPUF chips and devices using certified executions. The central chip would have CRPs for each of the computers it would be using, and perform computations using protocols similar to the one described in this section.

Chapter 6

Physically Obfuscated Keys

In trying to make a PUF, we have found ways of extracting information from complex physical systems, in a way that is reliable, and that resists attackers trying to extract the information through invasive means. In this chapter, we use the same techniques, but instead of using them to make a PUF, from which we can extract a limitless supply of responses, we choose to only extract one (or possibly a few responses).

6.1 Who Picks the Challenge?

The applications that we have been considering for now involve interaction between a PUF equipped device and a remote party who provides a challenge. There are applications in which that model isn't adequate. For example, a company might want to produce a chip that contains its latest video compression algorithm. They would like that chip to work right out of the box: connect an uncompressed video source to the input, turn on the power, and a stream of compressed video emerges from the output. However, they are concerned about people opening the chip, generating masks from what they see [5], and cheaply mass producing clones of the compressor.

PUFs as we have been using them so far are not suited to the problem at hand, as the chip isn't trying to engage in a cryptographic protocol with some remote party. However, by looking at PUFs in a new way, it is possible to get some interesting results. Here is a proposed solution to the company's problem, assuming that the chip is a microcontroller with the compression algorithm stored in ROM. It is illustrated in Figure 6-1.

First, the ROM is encrypted using a k-bit key K. To generate that key, A PUF is intertwined with the other functions on the chip. The PUF is hard-wired to always get the same challenge, and output a k-bit response. That response is combined with the contents of some fuses[1] via an exclusive-or operation to produce K. A decoder uses that K to decrypt the ROM.

Since cost constraints require that the same ROM be present on each chip, the key K must be the same for all chips. The response from the PUF is different for each chip, but

[1] A fuse is a memory element that can be set once. It is implemented by a fuse that can be burned out or not depending on the bit to be stored, hence the name.

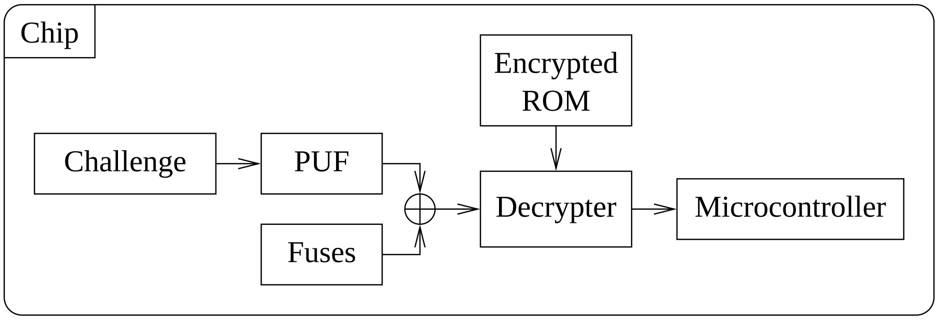


Figure 6-1: Using a PUF to generate a key

by setting the fuse bits appropriately for each chip, the key that the decoder gets is the one that is needed to decrypt the ROM. The fuse bits can be set while the chip is in testing, the manufacturer tells the chip K, and the chip automatically sets the fuse bits. This way, the response never has to leave the chip.

We have created exactly what we wanted. A chip that cannot be simply cloned. Indeed, techniques to extract the chip's layout [5] will be unable to deal with the PUF's physical system. So, even though the state of the fuses will be discovered, the value of K will remain secret, and copies of the chip that are made will not work.

What we have made is what we call a Physically Obfuscated Key. It is used exactly as a digital key would be, but it is more resistant to physical attack because it extracts its information from a complex physical system.

6.2 PUFs vs. POKs

The relation between PUFs and POKs is an interesting one. In Section 2.4.2, we mentioned POKs as a way of building PUFs. Now, we use PUF technology to produce a POK. From the technological point of view, they are actually quite similar, which explains why either one can be built from the other.

In this thesis, we have chosen to focus mainly on PUFs, but at first glance, POKs seem more powerful. Indeed, POKs directly solve the problem we started off from, which was to store a secret in a physical device. If we store a private key as a POK, nothing is to prevent us from using a classical public key infrastructure instead of the more complicated infrastructure that is described in Chapter 5. Moreover, POKs have many of the interesting properties of PUFs, such as detecting tampering that takes place while they powered down, without requiring any monitor circuitry to be permanently turned on.

Our reason for preferring PUFs is that we feel they are more secure. With POKs, all the secret information is present on the chip at once in digital form. An attack that manages to capture that information completely breaks the POK. To completely break a PUF device requires complete cloning, as an attacker who has momentary access to the PUF doesn't know which CRPs he should capture. Likewise, an attacker who manages to capture the secrets that are on the device at some time, only compromises the current transaction, and the transactions whose security is based on secrets that were on the device at the time of the compromise. Unrelated transactions, for example ones that are carried out by a different user won't be compromised unless the adversary carries out an independent attack on them. It is this greater compartmentalization of PUFs that makes us prefer them over POKs.

6.3 Elements of POK Security

One major problem with POKs is that they are vulnerable to some attacks that plague digital keys. Indeed, the POK contains a lot of digital logic to process data coming from the underlying physical system. Consequently, the response that gets combined with the fuse bits has to be stored digitally, probably in some SRAM cells. After hours of use, always with the same key stored in the SRAM, the key value will end up being burned into the SRAM in a way that can be exploited by an invasive attacker [12]. In fact, a number of similar attacks exist [2]. Therefore, the use of a POK does not remove the need for careful design of the digital logic that uses the key.

As for PUFs, some simple techniques can be used to strengthen POKs so that they remain effective in the open-once model. Indeed, as we have described them so far, POKs are completely vulnerable to that type of attack. The idea, as for PUFs, is to use two secrets instead of one (see Section 5.2.5). We place two POKs in the device, only one of which is turned on at any given time. We split the task to be done into two parts, each of which requires only one of the keys. The device then operates by turning one POK on, doing one half of the work, turning that POK back off, and repeating the process with the other POK and the other half of the work. An adversary who opens the device now only gets half of the key.

Of course, if the adversary now opens another chip, he can hope to get the other half of the key. Therefore, we find that the task must be split in two in a different way on each instance of the device, except for applications which, unlike our video compressor, have a different K on each device. Splitting a task in two parts differently on different devices is not always easy. In the case of RSA it can be done by expressing the secret key as a product of two partial secret keys. Decrypting for the secret key is then equivalent to decrypting using one partial key, and then the other. An ample supply of such splits exist as almost any exponent can be used as the first key.

The big problem that is left for our video compression application is performance. In the single POK case, the decryption of ROM contents is likely to be the bottleneck of the system. In the open-once case, where two keys are used in alternation, things are even worse, as each decryption requires generating and then forgetting each half of the key, in addition to the usual encryption tasks. Needless to say that given the complexity of generating a key, getting data from the ROM will be very slow.

If the device is to have acceptable performance, some kind of caching is necessary to allow the processing core to go faster than the decryption process. Unfortunately, the cache is an ideal way to leak parts of the decrypted ROM contents to an invasive adversary. Moreover, since the decrypted ROM contents are the same on each device, compromising multiple devices allows an attacker to recover more and more of the information that we were trying to protect.

Clearly, the application we have presented here is not ideally suited to POK techniques, because of the difficulty in reaching a good tradeoff between performance and security. Perhaps other applications with less frequent demands on stored secrets would nevertheless be able to benefit from POKs. For example an encrypted CD could be equipped with a POK device that contains the decryption key. The key would only be made available to CD readers conforming to a suitable Digital Rights Management system.

Chapter 7

Conclusion

7.1 Future Work

This thesis has introduced the concept of Physical Random Functions, and has attempted to convince the reader that it is an achievable and useful concept. Our strategy has been to describe all the basic parts that are necessary to achieve a few simple applications. However, the number of parts being large, we haven't been able to explore each one of them in much detail. Consequently, there is a lot of interesting work still to be done, and we will try to point out some of the topics that we consider important.

- Improve on the rudimentary Error Correction Code algorithm that we have presented (see Section 4.5).

- Try to better understand the effects of environmental parameters on the delay circuits, to devise better compensation algorithms. Environmental variations are the dominant type of noise we have to deal with, and any reduction in noise would make the error correction easier, so this is an important topic for research.

- Test the intertwining methods that we have proposed. See how much the PUF actually changes when a device is opened and when tampering or probing is attempted.

- Build an actual Silicon PUF candidate and challenge researchers in the field to try to break it. The toughest test of a security device is to put it in the field where people will try to attack it. This method has been successfully applied for encryption algorithms and pseudo-random functions, and is probably the only way to get a real idea of how hard something is to break.

- Study other ways of measuring delays. If possible, devise a direct delay measurement method that, unlike the arbiter method, outputs the delay of the circuit.

- Consider other ways of making delay-based PUFs harder to model.

- Try to implement an optical CPUF.

7.2 Final Comments

To conclude this thesis, I would like to summarize the main ideas that it brings to light. Initially, PUFs are identical, except in name, to Physical One-Way Functions [21]. Instead of focusing on optical technology, we have focused on silicon technology, but the same basic idea is present: there is no general way to copy complex physical systems, so let us try to use them to identify a device.

The major new idea in our work is control. Without it, we could only identify physical systems. With it, we can additionally bind computation to a specific physical system. Instead of serving as an identifier for itself, the physical system identifies a processing element. The range of applications explodes because of this simple difference.

It is amusing to look at the interaction between the logical and the physical part of a CPUF. The physical part identifies the logical part, and protects it from invasive attackers who would like to probe and tamper. In exchange, the logical part makes the physical part less vulnerable to model building attacks, and corrects the errors that are inherent in extracting information from a physical system. From a security point of view, the intertwined combination of those two elements is much more than the sum of the parts.

Turning a problem into an advantage is always a satisfying achievement. In this case, we have been able to put to use the manufacturing variations which, usually, only have negative effects in the silicon industry. It seems to often be the case that the security community takes people's problems and uses them to enable new security ideas.

There is still a lot of work to do to gauge exactly how strong PUFs are compared to classical primitives. What is certain is that as always in security, there is no absolutely unbreakable system. Our only expectation is that PUFs can help us reach a greater level of physical security than classical methods, at a lower cost.

Glossary

Additive Delay Model: A model of a circuit that assumes that the delay of a path through the circuit is the sum of the delays of the components and wires along that path.

Analog PUF: A PUF that is based on a complex physical system that doesn't involve any digital processing (we do not exclude digital components from analog PUFs, they are simply used there in an unconventional way).

Arbiter: Component that decides which of its inputs changed state first. If both inputs change state in too small a time interval, the output is arbitrary.

Birthday Paradox: When a set of cardinality n is sampled k times, the probability that the same element will have been picked twice approaches $\frac{1}{2}$ when $k^2 \approx n$. This phenomenon is called the birthday paradox because it is typically applied with surprising results to the probability of two people in a group having the same birthday.

Certified Execution: A certified execution protocol allows a protocol to run a program on a device, and get a certificate at the end of execution that proves that his program ran correctly on that device and produced the indicated results.

Characterize: The secret information in a PUF device is often determined by random manufacturing variations. We call the process of getting to know the PUF, usually by getting CRPs, characterization.

Classical Methods: Methods that store a secret in a device in digital form. Often barriers surround the secret so that an invasive adversary who tries to get at the secret will destroy it in his attempt.

Compensation: A technique to reduce dependence on environmental variation by performing two measurements and dividing them by each other. See Section 3.2.3 for details.

CRP: Challenge-Response Pair. A pair made up of an input to a PUF (challenge) and the corresponding output (response).

CPUF: Controlled Physical Random Function. See Section 2.1.2 for a detailed definition.

FPGA: Field Programmable Gate Array. A type of programmable logic device that contains a matrix of logic elements, which can be configured into arbitrary logic circuits.

IC: Integrated Circuit, also commonly referred to as a chip.

Intertwining: To protect the control logic of a CPUF from invasive attacks that attempt to eavesdrop or tamper with it, we intertwine the control logic with the physical system that defines the PUF. The rationale is that the adversary will change the PUF while tampering with the control logic, rendering his attack useless.

Keycard Application: The simplest application of a PUF. A card containing a PUF is used as a key. Section 2.2 details the application.

Manufacturer Resistance: A PUF that not even the manufacturer could make a clone of is said to be manufacturer resistant. See Section 2.1.3 for a detailed definition.

Open-Once Model: An attack model in which a physical adversary can open the device at a particular instant and get full access to the digital content (i.e., he can read all the bits in the device, and tamper with the digital functionality at will). However, as of that instant, the PUF ceases to function so all queries to the PUF will return bogus values. Details are in Section 2.3.3.

Path Through a Circuit: A sequence of wires and components in a circuit through which a signal can propagate from the circuit's input to its output.

Pre-challenge: A seed that is used in combination with a hash of a program to generate a challenge. See Section 5.2.2 for details.

POK: Physically Obfuscated Key. See Chapter 6.

PUF: Physical Random Function. See Section 2.1.1 for a detailed definition.

Smartcard: A card (often credit card sized) that contains a chip. Some typical applications are pay-phone cards, credit cards, keycards, or the SIM card in a cellular phone (a smaller card size is used in that case).

Vectorization: When making a strong PUF from a weak one, each query to the strong PUF results in multiple queries to the weak PUF. We call this technique vectorization. Section 4.2 covers this topic.

Weak PUF: A PUF that is somewhat predictable, and that suffers from output noise, but that we can strengthen using the methods in Chapter 4.

Bibliography

[1] Amer, Nabil Mahmoud, DiVincenso, David Peter, Gershenfeld, and Neil. Tamper detection using bulk multiple scattering. US Patent, number 5790025, August 1986.

[2] R. Anderson and M. Kuhn. Tamper Resistance - a Cautionary Note. In *Proceedings of the Second Usenix Workshop on Electronic Commerce*, pages 1–11, November 1996.

[3] Ross Anderson and Markus Kuhn. Low Cost Attacks on Tamper Resistant Devices. In *IWSP: International Workshop on Security Protocols, LNCS*, 1997.

[4] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley and Sons, 2001.

[5] S. Blythe, B. Fraboni, S. Lall, H. Ahmed, and U. de Riu. Layout reconstruction of complex silicon chips. *IEEE Journal of Solid-State Circuits*, 28(2):138 – 145, February 1993.

[6] D. S. Boning and S. Nassif. Models of Process Variations in Device and Interconnect. In W. Bowhill A. Chandrakasan and F. Fox, editors, *Design of High Performance Microprocessor Circuits*, chapter 6. IEEE Press, 2000.

[7] David Chaum. Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM*, 28:1030–1040, 1985.

[8] David Chinnery and Kurt Keutzer. *Closing the Gap Between ASIC & Custom*. Kulwer Academic Publishers, 2002.

[9] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Silicon physical random functions. In *Proceedings of the* 9^{th} *ACM Conference on Computer and Communications Security*, November 2002.

[10] Blaise Gassend, G. Edward Suh, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Caches and merkle trees for efficient memory authentication. In *Proceedings of Ninth International Symposium on High Performance Computer Architecture*, February 2003.

[11] Blaise Gassend, G. Edward Suh, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Caches and merkle trees for efficient memory authentication. In *Proceedings of Ninth International Symposium on High Performance Computer Architecture*, February 2003.

[12] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *The Sixth USENIX Security Symposium Proceedings*, pages 77–90. USENIX Association, 1996.

[13] R. Hamming. *Coding and Information Theory*. Prentice-Hall, Englewood Cliffs, 1980.

[14] Auguste Kerckhoffs. La cryptographie militaire. *Journal des Sciences Militaires*, pages 5 – 38, January 7^{th} 1883.

[15] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. *Lecture Notes in Computer Science*, 1666:388–397, 1999.

[16] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 169–177, November 2000.

[17] K. Lofstrom, W. R. Daasch, and D. Taylor. IC Identification Circuit Using Device Mismatch. In *Proceedings of ISSCC 2000*, pages 372–373, February 2000.

[18] Joe Loughry and David A. Umphress. Information leakage from optical emanations. *ACM Transactions on Information and System Security (TISSEC)*, 5(3):262–289, 2002.

[19] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[20] NIST. FIPS PUB 180-1: Secure Hash Standard, April 1995.

[21] P. S. Ravikanth. *Physical One-Way Functions*. PhD thesis, Massachusetts Institute of Technology, 2001.

[22] R. Rivest. RFC 1321: The MD5 Message-Digest Algorithm, April 1992. Status: INFORMATIONAL.

[23] S. H. Weingart S. W. Smith. Building a high-performance, programmable secure coprocessor. *Computer Networks (Special Issue on Computer Network Security.)*, 31:831–860, April 1999.

[24] Bruce Schneier. *Applied Cryptography*. Wiley, 1996.

[25] S. W. Smith and S. H. Weingart. Building a High-Performance, Programmable Secure Coprocessor. In *Computer Networks (Special Issue on Computer Network Security)*, volume 31, pages 831–860, April 1999.

[26] A. Thompson. An evolved circuit, intrinsic in silicon, entwined with physics. In Tetsuya Higuchi, Masaya Iwata, and L. Weixin, editors, *Proc. 1st Int. Conf. on Evolvable Systems (ICES'96)*, volume 1259 of *LNCS*, pages 390–405. Springer-Verlag, 1997.

[27] A. Thompson and P. Layzell. Evolution of robustness in an electronics design. In J. Miller, A. Thompson, P. Thomson, and T. Fogarty, editors, *Proc. 3rd Int. Conf. on Evolvable Systems (ICES2000): From biology to hardware*, volume 1801 of *LNCS*, pages 218–228. Springer-Verlag, 2000.

[28] H. Wong and Y. Taur. Three-dimensional atomistic simulation of discrete random dopant distribution effects in sub-0.1 um MOSFETs. In *IEDM Technical Digest*, pages 705–708, 1993.

[29] Bennet S. Yee. *Using Secure Coprocessors.* PhD thesis, Carnegie Mellon University, 1994.