2013

Physical Unclonable Function Techniques Applied for Digital Hardware Protection

Anthony Barrera
CUNY City College

Physical Unclonable Function Techniques Applied for Digital Hardware Protection

Thesis

Submitted in partial fulfillment of
the requirement for the degree

Master of Science, Computer Science

At

The City College of New York

of the

City University of New York

By

Anthony Barrera

September 2013

Approved:

Professor Izidor Gertner, Thesis Advisor

Professor Douglas R. Troeger, Chairman
Department of Computer Science

ABSTRACT

Privacy is an important property that is growing harder to keep as people develop new ways to steal information from users on their computers. Software alone cannot ensure privacy since an infected system is untrustworthy.

This paper presents several challenges malware brings that can be solved by using an external processor. Techniques such as keystroke encryption and message authentication can be used to protect users from having their passwords and other private data stolen.

To take advantage of the external hardware, a physical unclonable function can be used to generate private keys without the need for storing them in memory. In this report, a design of a physical unclonable function is detailed and designed for use on an FPGA. Two different types of hardware design software are briefly discussed for the purpose of choosing the superior tool for creating a PUF on an FPGA.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

Figure Page

GLOSSARY

Ammeter. An instrument that can determine the amount of electrical current being drawn.
ASIC. Application-Specific Integrated Circuit. An integrated circuit customized for a specific purpose.
FPGA. Field-Programmable Gate Array. An integrated circuit that can be configured by a user after the manufacturing of the circuit.
Keylogger. A software or device that can log everything that has been typed on a keyboard.
MAC. Message Authentication Code. Used to authenticate a message and check to see if it has been modified.
PUF. An acronym for Physical Unclonable Function.
Physical Unclonable Function. A function implemented on hardware that provides an unpredictable output to an input. ASICs with identical PUFs will give a different output to a same input.
Public-Key Cryptography. A cryptographic system that uses a pair of private and public keys. Both keys are needed to encrypt and decrypt data.
Symmetric-Key Algorithm. Cryptographic algorithms that use the same private key for encryption and decryption.

1. INTRODUCTION

As time passes, new software techniques are developed that make it more difficult to detect and remove malware from personal computers. A system that has had a rootkit installed can potentially steal data and never be discovered by antivirus software. The malware can be designed to deactivate and hide whenever the system is being scanned. Much more dangerous malware can attack and deactivate the protection that a system was using to ensure a user's protection.

Once a computer has been infected, it can no longer be trusted to handle private data. It may not even be possible to know if the system has been infected or not. It is for these reasons that software alone cannot perfectly protect a system and the privacy of any individual.

If we cannot trust antivirus software to protect our systems, then how can we ever safely and securely give our computers sensitive personal information? Personal information such as passwords, credit card numbers, social security numbers, and addresses can be stolen and used to steal identities or money stored in a bank. Everyone from an average person to a high-ranking government employee is at risk of losing data that other people should not get a hold of.

This document will focus on preventing one of the simplest ways that data can be stolen: keyloggers. Since a user's computer cannot be trusted to keep information perfectly secured, some of the responsibility for keeping information safe should be given to another device that cannot be controlled or tricked by an infected system.

Existing techniques and technologies will be explored to find a combination that could theoretically secure a computer from any keylogger threat. Such techniques include encryption, message authentication, and electrical current detection. Physical unclonable functions are explored as a way to improve the security of the hardware system. A simple design of a PUF is discussed and placed on an FPGA.

2. BACKGROUND INFORMATION

This section will discuss what a keylogger is and the types that exist. Generally, a keylogger can be in the form of hardware or software.

2.1. Software Keylogger

These keyloggers are programs that can capture what a user types using several different ways. There are 2 common types of software keyloggers: kernel-based keyloggers and application based keyloggers [1], [2]. An application keylogger can use the Windows hooking APIs to log what key has been pressed. Kernel-based keyloggers reside within the kernel and replace the driver that normally reads from the keyboard.

Both types of software keyloggers are installed as a result of something a user did [3]. The keylogger could have been installed by a malicious email attachment, a website script that exploits a browser vulnerability, or by other malware that may download and install the malicious software. These keyloggers all capture keystrokes and stores them into a file that may be sent over the Internet to the hacker at some point.

2.2. Hardware Keylogger

Hardware keyloggers are physical devices that must be installed on the system to log keystrokes. There are two categories of hardware keyloggers: external and internal [2]. Regardless of the type, a hardware keylogger requires physical access to the target computer in order to install and retrieve the device.

Since these keyloggers can passively monitor and log keystrokes, it is undetectable by software on a computer. The keyloggers have an internal memory that it uses to store

every keystroke. To access the data, a password must be typed in the keyboard that the keylogger is connected to. The keylogger will then start typing the captured data into the computer. By acting as a keyboard, the keylogger can display its memory contents without the need of software that would be detected by antivirus.

2.2.1. External Keyloggers

External hardware keyloggers are devices that are plugged inline with the keyboard and plugged directly into a port on the computer. Figure 2-1 shows an example of how an external keylogger would be connected to a keyboard and pc. Since it is out in the open, they can be spotted and removed. On a desktop PC, however, keyboards are most likely connected to a port on the back of the pc. Unless inspected regularly, they may never be noticed. This type of keylogger cannot be used on a laptop computer unless the user of the notebook regularly uses a dock with a separate keyboard.

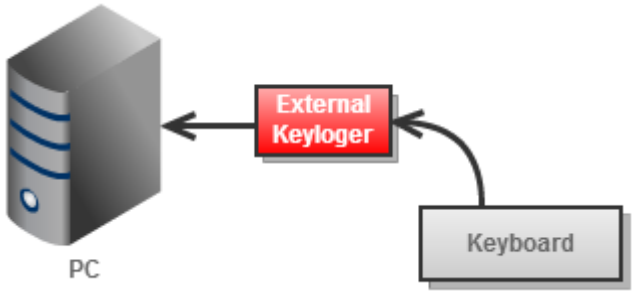

Figure 2-1: External Hardware Keylogger Connection

2.2.2. Internal Keyloggers

Internal keyloggers are circuits that are placed inside the pc or the keyboard. When placed within the PC, they are usually PCI cards that can be installed into an open slot of a desktop PC or notebook [2]. For keyboards, the internal keyloggers are circuit boards that are fitted into the existing circuitry of the keyboard. The keyboard must be

dismantled so that the keylogger can be installed. Keyboards with keyloggers built in can also be purchased.

The main advantage of using internal hardware keyloggers is that they will be hidden from plain view. It may be more difficult to install since it requires the disassembling of the target device to install the keylogger and to retrieve it. Rather than attempt to modify the keyboard, the keylogger user could swap out the target keyboard with an identical model that has the keylogger preinstalled. The same process can be used to retrieve the keylogger.

3. STATE OF THE ART IN KEYLOGGER PROTECTION

As keyloggers are capable of stealing information easily, there have been ways developed to help keep a user's privacy.

3.1. KeyScrambler

This software, developed by the QFX Software Corporation, protects users from software keyloggers by encrypting keystrokes [4]. The software encrypts keystrokes at the driver level, making it more difficult for software keyloggers to bypass the encryption. Any software keyloggers that are in the system and not in the kernel will only be able to capture the encrypted keystrokes. KeyScrambler decrypts the data once the keystroke arrives to the destination application.

The KeyScrambler software uses Blowfish for 128-bit symmetric-key encryption and RSA for 1024-bit asymmetric-key encryption. Unlike traditional antivirus, the software does not rely on knowing the signatures of keyloggers to stop information theft. This has the advantage of being able to defeat known and unknown software keyloggers in the system. Traditional antivirus would only be able to detect keyloggers that have been previously discovered and logged in a virus dictionary [5].

KeyScrambler's protection is active and will prevent a software keylogger from stealing unencrypted data. A traditional antivirus may not have detected a keylogger until data has already been stolen. This may be because the antivirus did not have a signature for the malware, or it did not detect suspicious behavior until it was too late.

3.2. Antivirus and Symantec Striker System

Symantec's Striker system aims to improve the way antivirus software detects malware. Traditional antivirus software has two tactics for detecting viruses: signature-based detection and behavior-based detection [5]. As was discussed in the previous section, signature-based detection is not very effective when it comes to detecting newly created viruses, since they may not have been added to a dictionary yet. Another problem arises when we consider the existence of polymorphic viruses. These viruses have the capability of encrypting every new infection, changing the signature of the virus, rendering the signature-based detection schemes useless [6].

Through behavior-based detection, antivirus software can detect suspicious activity and flag the program. Antivirus software can passively detect suspicious activity by scanning running programs and checking for deviations in activity. Antivirus software can actively detect suspicious behavior by executing a program within a virtual machine and checking to see if it matches criteria to be labeled malware. Behavior based detection allows antivirus software to detect any malware that matches its negative criteria, regardless if the malware has been added to a signature dictionary.

The Symantec Striker system improves upon the behavior-based detection by reducing the overhead of active scanning [6]. When running a program, instead of relying of heuristic guesses, it will examine behavior and rule out common malware behavior from a database. For example, if the scanned program infects EXE files, Striker will strike from the list any viruses that scan other file types. It will continue striking from the list

until it rules out all potential viruses. This method is faster than a normal heuristic-based system.

3.3. Dynamic Taint Analysis

This technique attempts to detect the existence of kernel-based keyloggers by tainting data [7]. Kernel-based keyloggers are capable of hiding their presence from antivirus software and are hard to detect. If tainted data has been modified, this can be detected and the cause of the modification can be found. If the cause of the modification is not from a legitimate source, then it is assumed to have been the cause of a malicious program.

4. PROBLEM STATEMENT

Though we may have effective antivirus and privacy software, can we ever be fully protected from keyloggers? Could we improve our protection by using hardware-based security?

4.1. Weaknesses of Current Technology

Though the KeyScrambler software seems to have a great solution to preventing data theft from keyloggers, there are vulnerabilities that exist that prevent it from being a perfect solution. Since KeyScrambler is software that resides in the kernel, any software keylogger that has managed to be installed within the kernel would not be affected by this protection. Furthermore, KeyScrambler cannot prevent hardware keyloggers from capturing keystrokes directly from the keyboard, as shown in Figure 4-1.

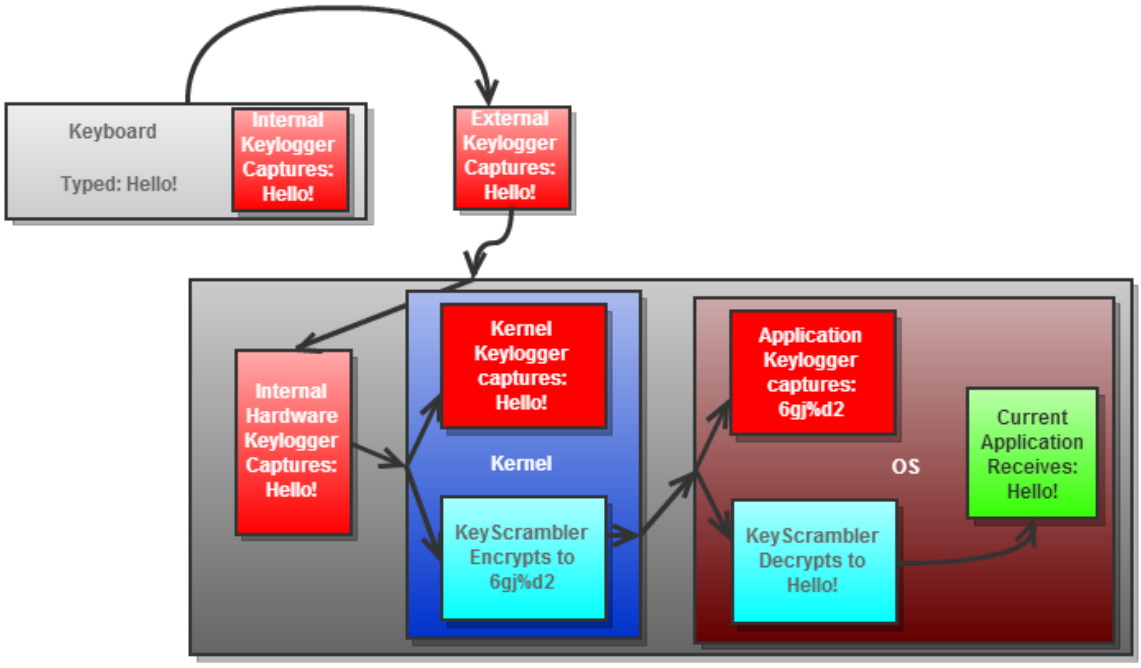

Figure 4-1: Keystroke Data Flow

Antivirus software can detect most malware, but hackers are learning new ways to hide malware within a system. Rootkits operate below the operating system and can control processes that are seen by the computer [8]. Malware with full control of a computer could corrupt the antivirus software to prevent it from being able to detect infections. One of the most powerful features of a rootkit is that it is capable of reinstalling itself from hidden area on the system after it has been removed.

Even with a more accurate detection of kernel-based keyloggers, as can be done using taint analysis, this does not prevent the problem of keyloggers. The researchers themselves question if a keylogger could be designed to bypass the taint analysis [7]. They mention that the taint detection could be broken if the keylogger uses other types of write operations.

4.2. Importance of Privacy

It goes without saying, that a great amount of damage could be caused if private sensitive information is stolen. In 2003, the source code for the unreleased video game Half Life 2 was stolen [9]. They stole enough of the data to be able to release a playable version within five days. The computers at the Valve game company had been infected with software keyloggers that their antivirus tools were unable to detect.

In another incident in 2003, JuJu Jiang installed software keyloggers at 13 different Kinko Internet café stores spread around Manhattan [10]. He managed to steal banking credentials of about 450 different people. With the information that he stole, he was able to create fake bank accounts and use them to transfer money from the real bank accounts that he stole the information from.

In 2011, two hardware keyloggers were found on computers in libraries located in Manchester [11]. Library staff that was inspecting a malfunctioning computer found the keyloggers. It is unknown who installed the hardware keyloggers or how long the scam has been going on for. The computers are protected with software that prevents people from installing software on the accounts of the machines. The hardware keyloggers was a work around to stealing data from people that use these public computers.

In 2013, malicious links have appeared on the popular website Facebook [12]. These links would lead to a website that would discretely install the Trojan horse named Zeus. This Zeus malware would use keyloggers to steal private login information for bank accounts. The malware gives very little warning signs to a user, such as crashing the pc, since its only purpose is to steal information. Facebook has suggested that users should take steps into protecting themselves from malicious links posted by others [13].

5. EXTERNAL HARDWARE SOLUTION

Various solutions to the problems brought up in section 4 are covered. Each of the following solutions may have their own possible problems that are also discussed and solved.

The main purpose of this report is to show how using an external piece of hardware can improve protection from any keylogger. Since the hardware is not a part of the computer, it cannot be modified by any malware that may be on the system, provided that the hardware was designed in a way that would not allow the system to alter functionality without a user's permission.

As shown in Figure 5-1, the hardware would be placed between the keyboard and the computer. The main purpose of the hardware would be to encrypt keystrokes a user would input and send the computer the encrypted keystroke that a keylogger would capture. The encrypted keystroke would then be decrypted on the pc and sent to the target application.

The following sections will explain in detail how the external hardware protection system will work, and how the system should be designed. For simplicity and clarity, the external hardware solution discussed will be referred to as the External Keylogger Barrier, or EKB.

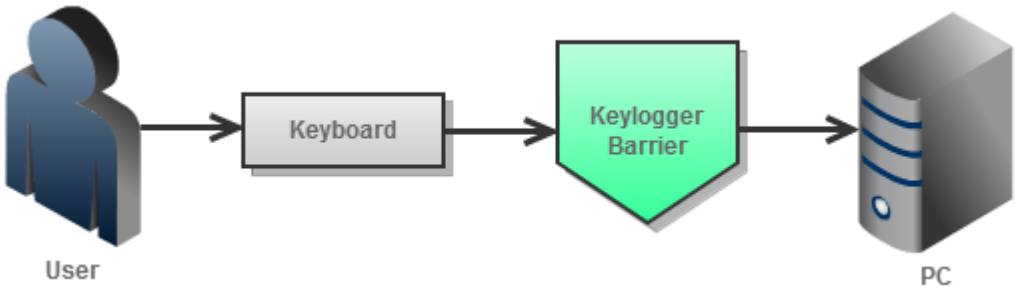

Figure 5-1: Diagram of External Hardware Protection Connection

5.1. Physical Unclonable Function (PUF)

One huge advantage to using a hardware solution is being able to use a physical unclonable function for cryptographic functions. Physical unclonable functions, or PUFs, are circuits that are capable of generating unique responses to inputs [14]. A PUF response is determined by the physical characteristics of the integrated circuit; two Identical PUFs will have unique outputs to the same input, since transistors will have slight differences in its inherent delay [15]. Traditionally, secret keys used for cryptographic functions are stored on nonvolatile memory within a chip that is vulnerable to invasive attacks [15]. A PUF does not need to store the key and can generate it's unique response at start up. The main problem with a PUF is the reliability of a stable output, but research has been done improving the quality of a PUF [14]–[16].

5.1.1. PUF Implementation

According to [15], an RO PUF (ring oscillator PUF) can be easily implemented on an ASIC (application-specific integrated circuit) or FPGA (field-programmable gate array). An RO PUF is based on delay loops and counters; the delay loop (ring oscillators) will have different delay times that's are determined by variations in the manufacturing of the transistors within a circuit.

An example of a ring oscillator can be seen in Figure 5-2. This ring oscillator was designed using the Quartus II design software. It has five stages of inversion that will create an oscillation and includes a NAND logic gate with an enable input to turn the oscillator off. Each oscillator will have a different frequency due to variations in the transistors within each of the logic gates.

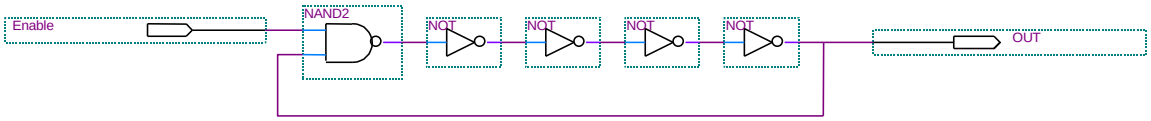

Figure 5-2: Five-Stage Ring Oscillator

Figure 5-3 shows a general design of an RO PUF. The RO PUF contains two multiplexers, two counters, a comparator circuit, and several ring oscillators. The RO PUF generates an output by comparing the frequencies of two oscillators by counting the amount of times it has oscillated. If the first oscillator causes is faster, the comparator will output a logical true, otherwise, a logical false.

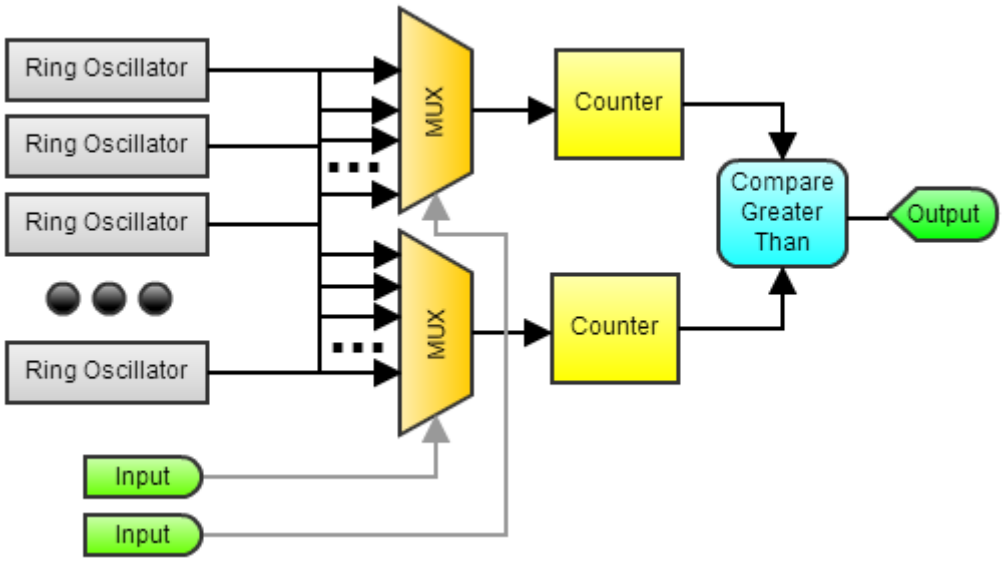

Figure 5-3: Ring Oscillator PUF

The RO PUF can generate different outputs depending on the input. The input goes into each of the multiplexers to select two of the oscillators. The oscillators chosen will increment the counter while it is enabled. When the oscillator is disabled, the counted value can be compared and give a logical true if the first counter has a higher value, i.e. the first oscillator has a higher oscillating frequency.

The RO PUF can be designed to cater to different cryptographic needs. The following sections will discuss how a PUF can be used to increase the security of the EKB. The next chapter will go in depth in how to implement a PUF into an FPGA.

5.2. Keystroke Encryption

Encryption is the EKB's main form of protection against hardware keyloggers. The process is shown in Figure 5-4. Any keystrokes input by the user will be encrypted on the EKB and sent to the computer as keystrokes. Any software keylogger on the system will capture the encrypted keystrokes and gain no information about what has been written. Secure software will then decrypt the keystroke and send the unencrypted data to the user's target application.

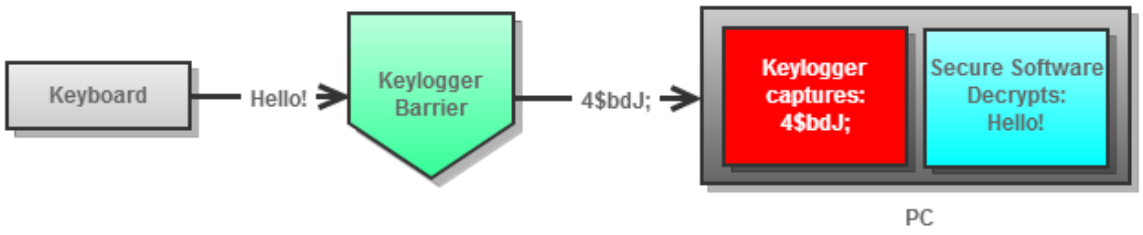

Figure 5-4: EKB Encrypting Keystrokes

The computer receives encrypted keystrokes as if the user was typing them. Any software keylogger would just record the encrypted keystrokes. The keylogger would not be able

to capture the unencrypted keystrokes, no matter how deep within the system it may be, since the keystrokes are encrypted by the EKB before they arrive to the computer.

5.2.1. Encryption Implementation

According to [17], symmetric-key algorithms can be implemented highly efficiently on hardware. This is because symmetric-key algorithms use table look-ups and bit-wise operations that can be parallelized to have higher performance than a software implementation of the same encryption algorithm.

Since the user's keystrokes are always being encrypted in this system, it is important for both the EKB and the software to quickly and efficiently encrypt/decrypt data. In order for a symmetric-key encryption to work, the decryption software on the system must have the key that was used to encrypt the data. Figure 5-5 shows how the decryption algorithm requires the same key that was used to encrypt the original plaintext data. In order to keep the system secure, the same key must not be used more than once; a new key must be generated each time a new session is started. The EKB and the decryption software must be able to share a key before they can start encrypting and decrypting keystrokes.

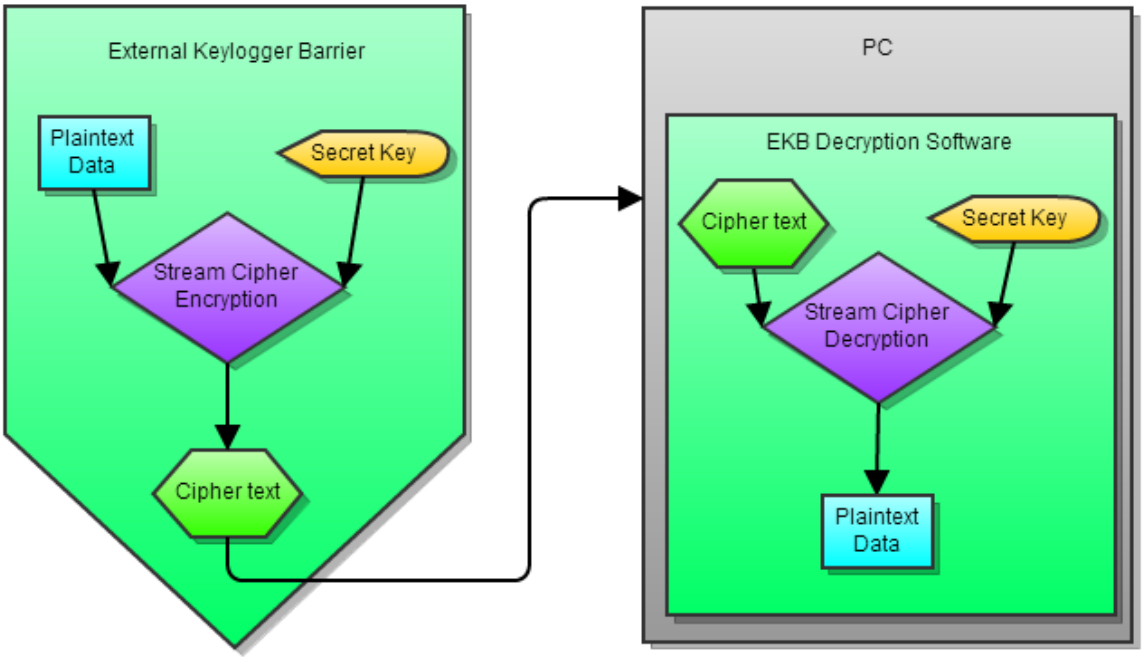

Figure 5-5: Example of Symmetric-Key Encryption

5.2.1.1. Key Sharing

The EKB and the decryption software must be capable of privately sharing a key to take advantage of symmetric-key encryption. An asymmetric-key algorithm can be used to encrypt the symmetric key and share it with the decryption software. Asymmetric-key algorithms work by creating a public key that is used to encrypt data that can only be decrypted using a private key. Since no prior shared secret is needed for decryption, this method can be used initially to securely share keys for the symmetric-key encryption.

Figure 5-6 shows a diagram of how public-key encryption generally works. The EKB would use its own private key in combination with the PC's public key to encrypt any data. The cipher text is then sent to the PC to be decrypted. The PC must use its own private key in combination with EKB's public key to decrypt the data. It is important that each of the public keys are verified to prevent a man-in-the-middle attack.

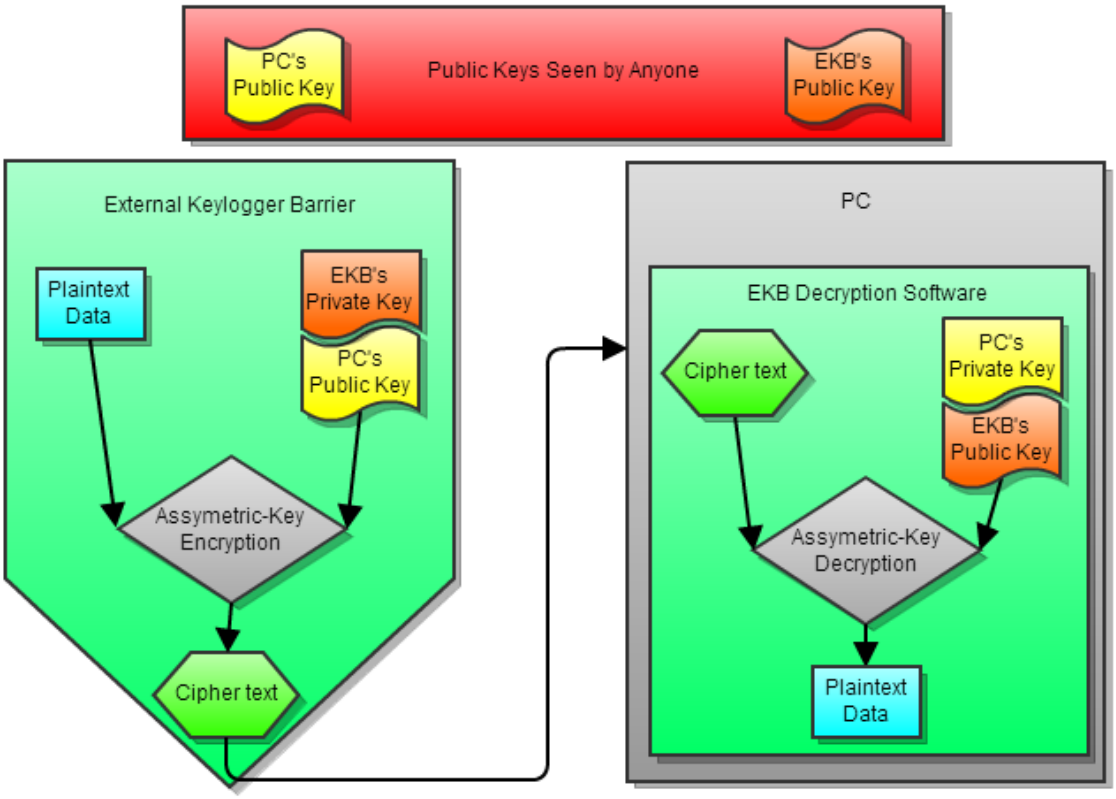

Figure 5-6: Public-Key Encryption Example

5.2.2. PUF Authentication

The PUF can be used to allow the PC to authenticate the EKB to prevent a man-in-the middle attack. In a public-key algorithm, this attack works by substituting the public keys with the attacker's keys so that the attacker is able to decrypt any cipher text. The attacker can then encrypt the text back using the original public keys so that the presence of the attacker is not discovered. To prevent this, a MAC (message authentication code) can be used to verify that the PC is communicating with the EKB.

As mentioned in section 5.1.1, a PUF will generate a unique output on different ASICs, and it is not possible to predict the output response of a PUF for any input. Using this property, it is possible to authenticate a device by giving the PUF a challenge input and

checking to see is the output matches a stored result [15]. Figure 5-7 shows a diagram of how the PUF output can be used in a MAC algorithm. Since a man-in-the-middle cannot predict a PUF response to a challenge, the attacker cannot modify EKB's public key without invalidating the MAC; the attacker cannot create a new MAC since the PUF response is not known. The software on the PC checks to see if the key has been modified by generating the MAC and checking to see if it matches the received MAC.

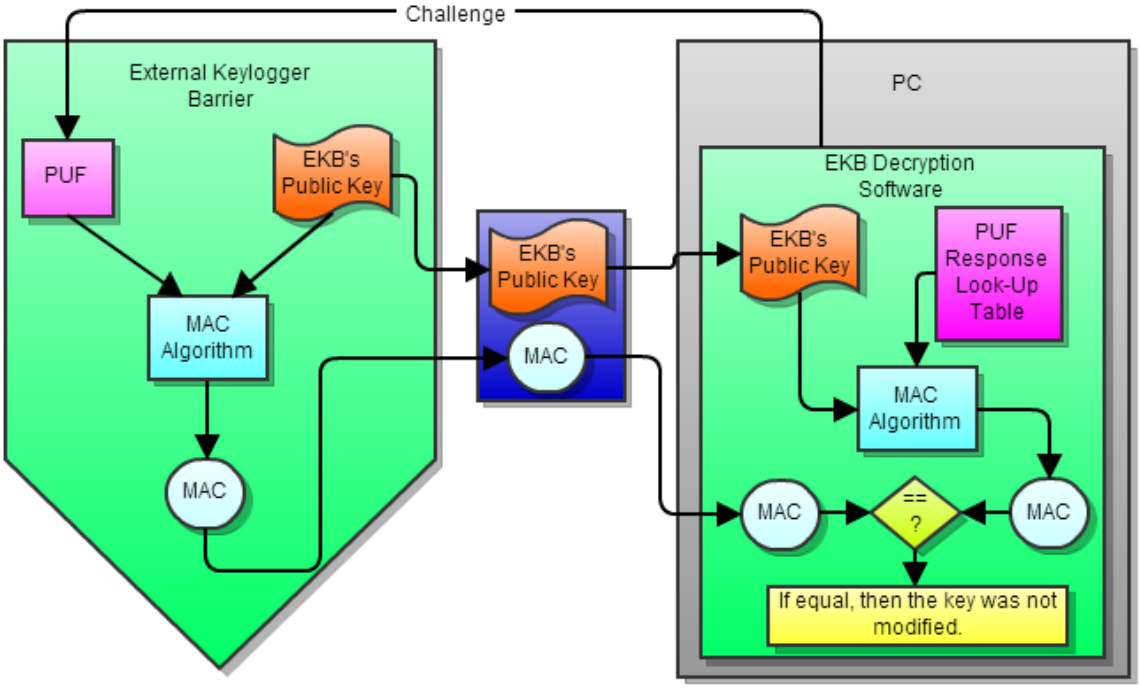

Figure 5-7: PUF Message Authentication

Once the public keys have been authenticated, the public-key algorithm can be used to share the private keys for symmetric-key encryption. The EKB can also send the software new PUF challenge and response pairs for future authentication.

5.3. Hardware Keylogger Detection

Encrypting keystrokes will protect data from being stolen by software keyloggers, but hardware keyloggers require another solution. Figure 5-8 shows that the effectiveness of

keystroke encryption depends on the placement of the hardware keylogger. If the keylogger were connected between the pc and the EKB, then the keylogger would only capture encrypted data. If the hardware keylogger were placed between the keyboard and the EKB, then it would be able to capture the unencrypted data.

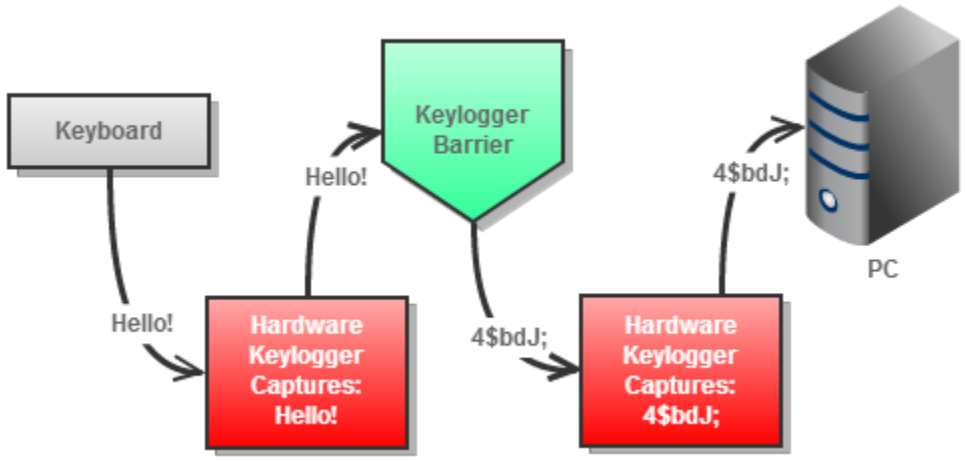

Figure 5-8: Keystroke Encryption Effectiveness Against Hardware Keyloggers

Though the vulnerability shown in Figure 5-8 exists, there is the advantage that the hardware keylogger would be easier to spot since it would not be hidden behind the pc, where the keyboard is likely to be connected. There are techniques that can be implemented on the EKB to detect the presence of a hardware keylogger.

5.3.1. Electrical Current Change Detection

In order for the hardware keylogger to be able to log any data, it must be connected in line with the keyboard. Connecting the keylogger to the keyboard can cause a noticeable change in the amount of electrical current drawn, since there is now additional hardware to power [18], [19]. Since software within a computer has no way of detecting a current change, this method of detecting hardware keyloggers cannot be used.

An ammeter is an instrument that can measure the electrical current in a circuit. If the EKB had an ammeter, it would be able to detect any changes made to the keyboard setup by comparing the amount of electrical current the keyboard is pulling to the amount of current that the keyboard normally pulls. If the amount of current drawn by the keyboard is higher than is normally expected of that keyboard, it should raise a flag and alert the user of a suspicious device.

5.3.2. Signal Propagation Time Increase Detection

Another way to detect a hardware keylogger is to detect an increase in the signal propagation time. The microprocessor within the keylogger needs to process the signal from the keyboard, and will make a noticeable propagation delay [18]. Since the delay created by the hardware keylogger is very small, a computer wouldn't be able to measure the delay accurately enough unless the detection program had exclusive access to the CPU [18].

The EKB can be designed to check the signal propagation time by pinging the keyboard. The normal propagation time for the keyboard can be stored for later use. If the EKB detects a greater propagation time, compared to the previously stored value, it should throw a flag and notify the user of suspicious hardware.

6. FPGA IMPLEMENTATION OF A PUF

While an RO PUF can be easily implemented into an FPGA, there are some strict requirements that need to be placed in order for it to work correctly [14]–[16], [20]–[22]. This chapter will cover the process of creating a PUF that is capable of working on an FPGA device.

6.1. RO PUF Design Considerations

As was stated in earlier in section 5.1.1, on page 13, a ring oscillator PUF is easy to implement on an FPGA. The reason RO PUFs are easier to implement is because there is only a single constraint that must be followed; every ring oscillator needs to be identically laid out [15]. Other types of PUFs, such as the arbiter PUF and butterfly PUF, require all interconnections between the gates to be identical [15], [20]. This constraint is what helped to decide which software was used to implement the PUF.

6.1.1. Choosing the Design Software

Altera Quartus II was the first software that was considered for designing the PUF for an FPGA. This is where problems occurred. After testing the RO PUF on multiple Altera brand FPGAs, the PUF did not give unique outputs. The RO PUF derives its unique response from the random variations from manufacturing that would cause different oscillation frequencies in each ring oscillator.

6.1.1.1. Understanding Ring Oscillator Delay

To understand why the Altera Quartus II software was not effective, the basic idea of how the RO PUF works will be explained. RO PUFs take two of the ring oscillators and

compares the frequencies. The frequency is determined by the delay throughout the oscillator circuit.

$$d_N = d_S + d_R \tag{1}$$

Equation 1 shows the delay, d_N, that is present in a ring oscillator [20]. Variable d_S is the static delay caused by the routing of the gates, while d_R is the random delay caused by variations in the manufacturing of the transistors within the logic gates.

$$\Delta d_R = d_{R1} - d_{R2} \tag{2}$$

$$\Delta d_S = d_{S1} - d_{S2} \tag{3}$$

$$\Delta d = d_{S1} - d_{S2} + d_{R1} - d_{R2} = \Delta d_S + \Delta d_R \tag{4}$$

Equation 2 shows the difference between the random delay of two different oscillators, and equation 3 shows the difference in the static routing delay between the same two oscillators. Equation 4 is the difference between the total delays of the two oscillators.

For the RO PUF to have a unique response in every ASIC, the ring oscillator frequency must be determined by random delays [20]. To achieve this, every RO must be laid out identically. With identical layout, the static delays can be assumed to be equal in every RO.

$$\Delta d = d_S - d_S + d_{R1} - d_{R2} = \Delta d_R \tag{5}$$

With identical static delays, the RO PUF result can be determined by just the random delays, as shown in equation 5. If the static delays are not equal, then the RO PUF results will be partially dependant on the static delays [20]. If the static delays are greater than the random delays, the result of all PUFs will be static and prevent unique response.

6.1.1.2. Problem with Altera Quartus II

Testing the RO PUF on FPGA boards showed that there was not enough random delay to provide unique PUF outputs. The ring oscillators had not been identically laid out prior to the tests.

The major problem with Quartus II was the lack of tools to route out the oscillators identically. According to [22], there is no function in the software to define a layout that can be applied to every oscillator. The user must individually and manually route every ring oscillator. Figure 6-1 shows why this is infeasible for even small RO PUF designs. The PUF shown contains 128 ring oscillators that would have to be manually routed to achieve the unique results of a PUF.

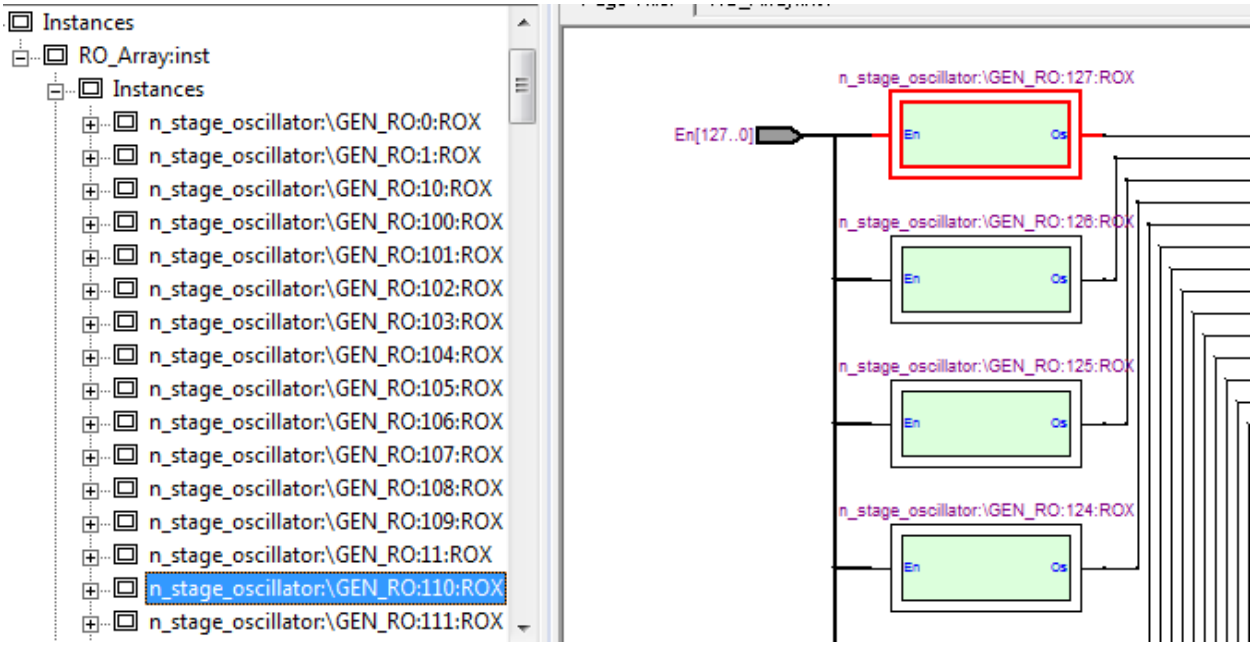

Figure 6-1: Quartus RTL View of RO PUF with 128 Ring Oscillators

6.1.1.3. Choosing Xilinx ISE

Xilinx is a common choice for implementing the RO PUF. The Xilinx software has a feature called hard-macro that allows a user to create a logical function that has been defined from components of a specific device family. This feature was used by [14]–[16], [20] to create a hard-macro of a ring oscillator to ensure that they are all identical. This single feature made Xilinx a perfect choice to design a PUF for.

6.1.2. General RO PUF Design

For the purposes of this paper, a simple RO PUF was designed. Figure 5-3, from page 14, presented the design of a single RO PUF. To simplify the circuit for testing purposes, the RO PUF was placed multiple times in parallel to have multiple output bits. The FPGA used was the Nexys 3. Since the Nexys 3 has 8 LEDs, eight PUFs were placed in parallel to create an eight-bit output. This larger output size made testing for uniqueness easier.

6.2. Designing the RO PUF

As mentioned in section 6.1.2, the Nexys 3 FPGA was used to test the RO PUF. A hard-macro of a ring oscillator was created to create identical layouts. Figure 6-2 shows a five-stage ring oscillator that was created into a hard-macro. This macro was used in VHDL code as a component to create the RO PUF. Figure 6-3 shows every component that went into the VHDL version of the design from Figure 5-3.

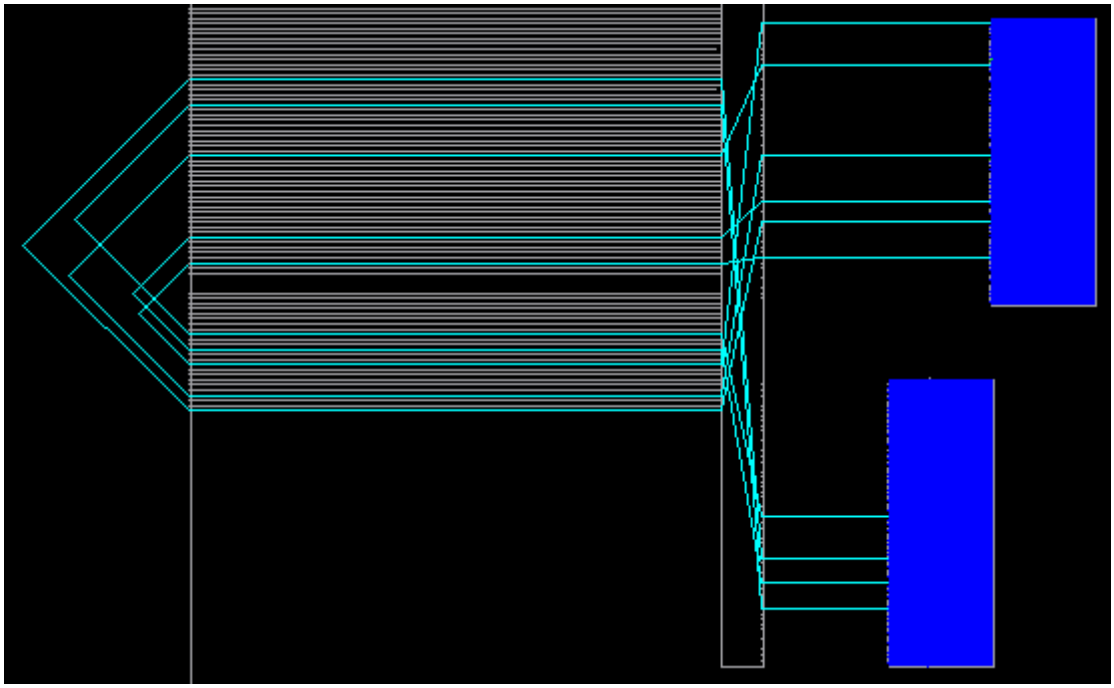

Figure 6-2: Five-Stage Ring Oscillator Hard Macro in Xilinx FPGA Editor

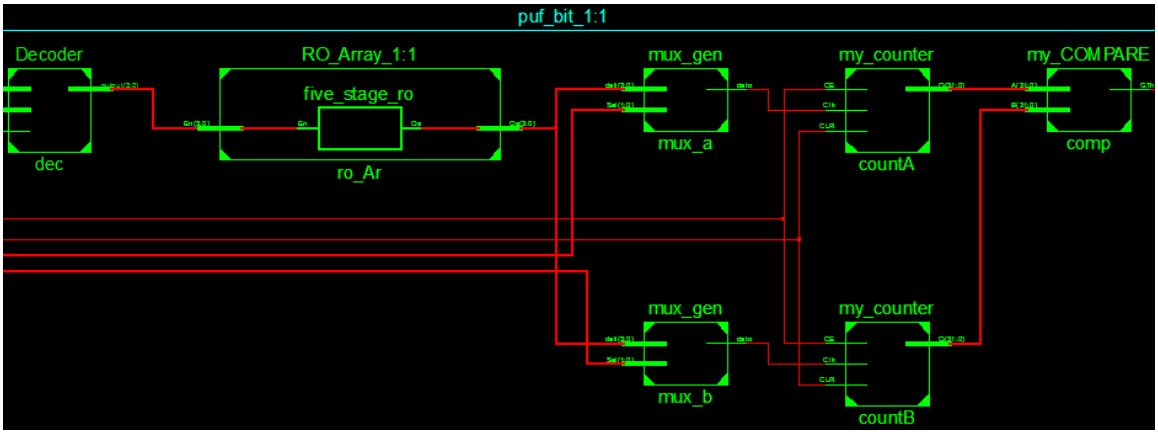

Figure 6-3: RTL View in Xilinx of single RO PUF

6.2.1. Controlling the RO PUF

The RO PUF on its own cannot create a useful output unless there's a state machine to control the functions of the circuit. Figure 6-4 shows the top level of the RO PUF that includes a state machine. This state machine controlled several parts of the PUF. The

state machine can make the counter reset its values, and enable or disable to incrementing of the counter. The state machine also enabled the ring oscillators for several clock cycles to allow the counters to accumulate oscillations from the two oscillators being compared.

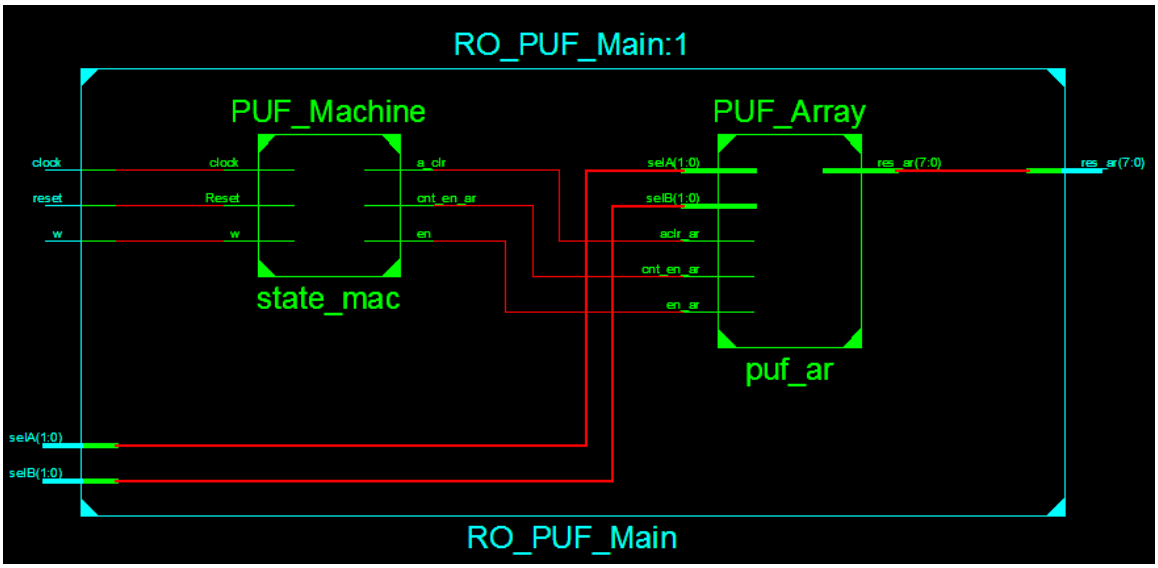

Figure 6-4: Top Level of RO PUF

6.2.2. Hard-Macro Placements

Hard-macros require a user to place each macro to a location on the FPGA. To save time, this process was automated by using VHDL.

```
for I in 0 to RO_Width-1 generate
     ---Sets the location of RO on the Nexys 3 FPGA
     attribute LOC of ROX : label is "SLICE_X"&INTEGER'image(ro_row *
4)&"Y"& INTEGER'image(I+9);
```

In the above code snippet, for each RO generated, a different location is assigned to the macro. The above values are specific to the Nexys 3 FPGA and the five-stage RO hard-macro.

6.3. Testing on FPGA

The RO PUF that was tested was a small circuit. There were 8 PUFs to give an 8-bit output. The 8 bits of the output were assigned to each of the LEDs on the Nexys 3 board. The RO PUF was tested on 2 separate FPGAs and had different results, which proved that the RO PUF could work on an FPGA. All of the VHDL files necessary to compile this PUF are in appendix A.

The best way to test the PUF for uniqueness is to take multiple FPGAs and give each board identical inputs and checking to see if the boards give unique outputs. If the results are matching, then there is too much static array and reduces the effectiveness of the random delays.

7. CONCLUSION

Software keyloggers and hardware keyloggers have been used to steal sensitive data such as passwords and credit card numbers. Keyloggers can be very hard to detect even by the best antivirus software, leaving many people vulnerable to loss of privacy. A better solution to the problem of keyloggers is to prevent keyloggers from getting your data in the first place.

Encrypting keystrokes prevents both software and hardware keyloggers from stealing the plaintext data. By encrypting the keystrokes from outside of the pc, it impossible for keyloggers to steal the plaintext keystrokes without the private key used to decrypt the cipher text.

Hardware keyloggers will also not be able to steal plaintext keystrokes if it is connected to the output of the keystroke encryption device. If it is placed before the encryption device, it will most likely be easier for a user to discover its existence in the system and be removed. Several techniques for detecting the hardware keylogger can also be employed. A hardware keylogger will increase the amount of electrical current drawn by the keyboard. An ammeter would be able to detect the current change and alert the user that a keylogger is on the system and should be immediately removed. A hardware keylogger can also cause a delay in the time it takes for a keystroke to propagate. By timing the amount of time it took for a keystroke to arrive, it can determine if there was a delay caused by a keylogger.

Using a hardware solution to keyloggers brings many benefits that a software solution alone would not bring. Symmetric-key algorithms can be implemented highly efficiently on hardware by parallelizing it. A PUF can be used to authenticate anything the device sends to the PC to prevent the communications from being intercepted by a man-in-the-middle attack. The threat of hardware keyloggers can be eliminated through the use of techniques that cannot be implemented by software.

7.1. Weaknesses

There are several weaknesses that a hardware solution may have. On the boot up of a pc, the software for decrypting keystrokes may not be active until the system has loaded. This may allow a kernel level keylogger to steal the login password before the device is able to encrypt data. Another weakness is the security of the private keys. A new private key is generated for each session, but if the private key is stolen from the memory of the PC, the encrypted data may be able to be decrypted.

7.2. Future Research

Some research into how to protect the software from being targeted by malware should be looked into. Some malware use polymorphic code to avoid detection from antivirus software. Perhaps polymorphism can be implemented to protect software from being modified by malware. By constantly changing the way the code runs, a virus may not be able to find the private key within the memory of the system.

7.2.1. Executable Polymorphism

Having the executable of the software reside on the external device to prevent modifications by a virus could increase the security of the system. The device could be

designed to modify the executable so that it is different each time it is run on the system. This could prevent a virus from being able to target certain memory addresses and inject instructions or steal private keys. A PUF could be used to digitally sign the executable to check if the code has been modified after having been loaded into the system's memory.

7.2.2. PUF Reliability

If a PUF is used for digital signatures, the PUF must be able to reliably generate a constant private key for cryptographic functions. A PUF output can be affected by the temperature or by the input voltage. Research must be done to allow a PUF to give a stable result under any operating conditions.

In the case of a RO PUF, a possible solution is to determine which ring oscillator pairs are more likely to cause an unstable result. This can be determined by checking the oscillation frequencies under normal conditions. If the frequencies are very close, then it should be considered as a possible unstable pair. This should be recorded on volatile memory with the result stored. Next time a PUF output is requested, the volatile memory should be checked for any unstable pairs used. If there is a match, then the result stored in memory should be used in place of the less reliable puff output.

8. APENDIX A: SOURCE CODE

8.1. Ring Oscillator Array

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

---RO_Width: Choose how many RO there are
---RO_Row:Used by upper level entities to make
----------the automation of attributing locations to
----------each RO easier.

---En: Enables the RO to oscillate.
---OS: The output of the RO.
entity RO_Array is
  generic (RO_Width : integer := 64;
              ro_row : integer :=1);
  port( En: in std_logic_vector(RO_Width-1 downto 0);
        Os: out std_logic_vector(RO_Width-1 downto 0));

end RO_Array;

architecture block_diagram of RO_Array is

    component five_stage_ro
      port(En : in  std_logic;
           Os : out std_logic);
    end component;

    attribute LOC : string;

    begin

    GEN_RO:
    for I in 0 to RO_Width-1 generate
     ---Sets the location of RO on the Nexys 3 FPGA
     attribute LOC of ROX : label is "SLICE_X"&INTEGER'image(ro_row *
4)&"Y"& INTEGER'image(I+9);
     begin
      ROX : five_stage_ro
        port map(En(I), Os(I));
    end generate GEN_RO;

end block_diagram;
```

8.2. Decoder

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;

----Decoder is used to activate specific RO for PUF
entity Decoder is
  ----The width is how many bits will be needed to address all RO
  ----Should be set to log_2(number of RO)
  generic (Sel_Width : integer := 8);
  port( en: in std_logic;
        inA, inB: in std_logic_vector(Sel_Width-1 downto 0);
        output: out std_logic_vector(2**Sel_Width-1 downto 0));

end Decoder;

architecture block_diagram of Decoder is
begin
process(inA, inB, en)
begin
   ---Set output bits to 0 to deavtivate all RO
    output <= (others => '0'); -- default

    ---activate RO corrspoending to inA
    output(to_integer(unsigned(inA))) <= en;

    ---activate RO corresponding to inB
    output(to_integer(unsigned(inB))) <= en;
end process;
end block_diagram;
```

8.3. PUF Bit

```
----- CELL Compare -----
--entity that does bit comparisons

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.std_logic_unsigned.all;

entity my_COMPARE is
port(
    A, B   : in std_logic_vector(31 downto 0);
    GTh  : out std_logic
  );
end my_COMPARE;

architecture arch_comp of my_COMPARE is
begin

  GTh <= '1' when ( A > B ) else '0';
```

```
end arch_comp;


----- CELL Counter -----
--entity that counts the oscillations of RO

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity my_counter is
port (
    Clk   : in STD_LOGIC;
    CE  : in STD_LOGIC;
    CLR : in STD_LOGIC;
    Q   : out STD_LOGIC_VECTOR(31 downto 0)
    );
end my_counter;

architecture arch_count of my_counter is

  signal COUNT : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');

begin

process(Clk, CLR)
begin
  if (CLR='1') then
    COUNT <= (others => '0');
  elsif (Clk'event and Clk = '1') then
    if (CE='1') then
      COUNT <= COUNT+1;
    end if;
  end if;
end process;

Q   <= COUNT;

end arch_count;


-----Cell MUX ------------------------------
--entity that is a mux for the PUF
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

ENTITY  mux_gen IS
    GENERIC (Sel_Size : INTEGER :=8);
   PORT (Sel         : IN  STD_LOGIC_Vector(Sel_Size-1 downto 0);
         dati        : IN  STD_LOGIC_Vector(2**Sel_Size-1 downto 0);
         dato        : OUT STD_LOGIC
        );
END mux_gen;
```

```
ARCHITECTURE archmux OF mux_gen IS

BEGIN

    dato <= dati(conv_integer(sel));

END archmux;

----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
----Main Code for Puf Bit-------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

---Sel_size_p:The width of the selector for mux and decoder
---row:used by upper level entity to assign this PUF a row number
-------to allow for easier attribution of location to each RO on FPGA

---en: Enables the RO
---selA: Select an RO
---selB: Select another RO for comparison to the first
---aclr: Clears counter that counts oscillations
---cnt_en: enables the counter.
---res: The result of the comparisons between 2 counters. 1 if RO A> RO
B
entity puf_bit is
    GENERIC (Sel_Size_p : INTEGER :=8;
                  row : INTEGER :=1);
    Port ( en : in  STD_LOGIC;
           selA : in  STD_LOGIC_VECTOR (Sel_Size_p-1 downto 0);
           selB : in  STD_LOGIC_VECTOR (Sel_Size_p-1 downto 0);
           aclr : in  STD_LOGIC;
           cnt_en : in  STD_LOGIC;
           res : out  STD_LOGIC);
end puf_bit;

architecture Behavioral of puf_bit is


component my_counter is
port (
    Clk   : in STD_LOGIC;
    CE    : in STD_LOGIC;
    CLR   : in STD_LOGIC;
    Q     : out STD_LOGIC_VECTOR(31 downto 0)
    );
end component;

component  mux_gen IS
    GENERIC (Sel_Size : INTEGER :=8);
```

```
   PORT (Sel          : IN  STD_LOGIC_Vector(Sel_Size-1 downto 0);
         dati         : IN  STD_LOGIC_Vector(2**Sel_Size-1 downto 0);
         dato         : OUT STD_LOGIC
        );
END component;

component Decoder is
  generic (Sel_Width : integer := 8);
  port( en: in std_logic;
        inA, inB: in std_logic_vector(Sel_Width-1 downto 0);
        output: out std_logic_vector(2**Sel_Width-1 downto 0));

end component;

component RO_Array is
  generic (RO_Width : integer := 64;
              ro_row: integer :=1 );
  port( En: in std_logic_vector(RO_Width-1 downto 0);
        Os: out std_logic_vector(RO_Width-1 downto 0));

end component;


component my_COMPARE is
port(
    A, B   : in std_logic_vector(31 downto 0);
    GTh  : out std_logic
  );
end component;

signal decout : std_logic_vector(2**Sel_size_p-1 downto 0);
signal RO_out : std_logic_vector(2**Sel_size_p-1 downto 0);
signal cntA, cntB : std_logic_vector(31 downto 0);
signal muxa,muxb : std_logic;

begin

dec :decoder GENERIC MAP(Sel_Width => Sel_size_p)
            PORT MAP(en=>en,
                     inA=>selA,
                     inB=>selB,
                     output=>decout);

ro_Ar :RO_Array GENERIC MAP(RO_Width => 2**Sel_size_p,
                                    ro_row=>row)
            PORT MAP(en=>decout,
                     os=>RO_out);

mux_a :mux_gen GENERIC MAP(sel_size => Sel_size_p)
            PORT MAP(sel=>selA,
                     dati=>RO_out,
                     dato=>muxa);

mux_b :mux_gen GENERIC MAP(sel_size => Sel_size_p)
            PORT MAP(sel=>selB,
                     dati=>RO_out,
```

```
                        dato=>muxb);

countA: my_counter PORT MAP(
                        Clk=>muxa,
                        CE=>cnt_en,
                        CLR=>aclr,
                        Q=>cntA);

countB :my_counter PORT MAP(
                        Clk=>muxb,
                        CE=>cnt_en,
                        CLR=>aclr,
                        Q=>cntB);

comp :my_COMPARE PORT MAP(
                        A=>cntA,
                        B=>cntB,
                        Gth=>res);


end Behavioral;
```

8.4. PUF Array

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

---puf_width: The amount of PUFs to have in parallel
---sel_width_ar:The select width for the mux and decoders in PUF


---en_ar: Enables the RO
---aclr_ar: Clears counters that counts oscillations
---cnt_en_ar: enables the counters.
---selA: Select an RO
---selB: Select another RO for comparison to the first
---res_ar: The result of the comparisons between 2 counters. 1 if RO A>
RO B
entity PUF_Array is
  generic (puf_Width : integer := 8;
           sel_width_ar : integer := 8);
  port( en_ar, aclr_ar, cnt_en_ar: in std_logic;
        selA : IN STD_LOGIC_VECTOR(sel_width_ar - 1 DOWNTO 0);
        selB : IN STD_LOGIC_VECTOR(sel_width_ar - 1 DOWNTO 0);
        res_ar: out std_logic_vector(puf_Width - 1 downto 0));

end PUF_Array;

architecture block_diagram of PUF_Array is

    COMPONENT puf_bit
    GENERIC (Sel_Size_p : INTEGER;
                 row : INTEGER
            );
    PORT(   en : in  STD_LOGIC;
```

```
            selA : in  STD_LOGIC_VECTOR (Sel_Size_p-1 downto 0);
            selB : in  STD_LOGIC_VECTOR (Sel_Size_p-1 downto 0);
            aclr : in  STD_LOGIC;
            cnt_en : in  STD_LOGIC;
            res : out  STD_LOGIC);
    END COMPONENT;

    begin

    GEN_puf:
    for I in 0 to puf_Width-1 generate
      pufX :puf_bit
            GENERIC MAP(Sel_Size_p  => sel_width_ar,
                                row =>I)
            PORT MAP(
                     en => en_ar,
                     selA => selA,
                     selB => selB,
                     aclr => aclr_ar,
                     cnt_en => cnt_en_ar,
                     Res => res_ar(I));
    end generate GEN_puf;

end block_diagram;
```

8.5. PUF State Machine

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

---delay: The amount of clock cycles to leave the RO enabled for.

---clock: a clock for this state machine
---w: Allows the state machine to leave the start state when high
---reset:resets the state machine to the start state
---en:output that enable RO
---a_clr:output that clears PUF counters
---cnt_en_ar:output that enables PUF counters to increment
ENTITY PUF_Machine IS
  generic (delay : integer := 100);
  PORT ( clock, w, Reset      : in std_logic;
           en, a_clr, cnt_en_ar : out std_logic);
END PUF_Machine;


ARCHITECTURE Behavior OF PUF_Machine IS
  TYPE State_type IS (start, clear, enable, disable);
  SIGNAL current_state, next_state : State_type :=start;
  SIGNAL counter : integer range 0 to delay-1 := 0;
  SIGNAL cnt_reset, cnt_en : std_logic;

BEGIN

  PROCESS (w, current_state, counter) -- state table
  BEGIN
```

```
        case current_state IS
            WHEN start => --wait here for w input of 1
                IF (w = '0') THEN
                    next_state <= start;
                ELSif(w = '1') then
                    next_state <= clear;
                END IF;
            WHEN clear => --clear the counters in the RO PUF
                next_state <= enable;
            WHEN enable => --This is a delay state to allow PUF to count
                if (counter = delay-1) then
                    next_state <= disable;
                else
                    next_state <= enable;
                end if;
            WHEN disable => --disable PUF and wait for w input to be 0
                IF (w = '1') THEN
                    next_state <= disable;
                ELSif(w = '0') then
                    next_state <= start;
                END IF;
        END CASE;
    END PROCESS; -- state table


    PROCESS (Clock, reset) -- state flip-flops
    BEGIN
        if (reset='1') then
            current_state<=start;
        elsif (rising_edge(clock)) then
            current_state<=next_state;
        end if;
    END PROCESS;

    PROCESS (Clock, cnt_reset, cnt_en, counter) -- state flip-flops
    BEGIN
        if (cnt_reset='1') then
            counter <= 0;
        elsif (rising_edge(clock) and cnt_en = '1' and counter /= delay-
1) then
            counter <= counter + 1;
        end if;
    END PROCESS;

    PROCESS (current_state) -- assign output values
    BEGIN
        case current_state IS
            WHEN start =>
                en<='0';
                a_clr<='0';
                cnt_en_ar<='0';
                cnt_reset<='1';
                cnt_en<='0';
            WHEN clear => --set clear up
                en<='0';
                a_clr<='1';
```

```
                cnt_en_ar<='0';
                cnt_reset<='1';
                cnt_en<='0';
            WHEN enable => --set enable up; reset clear down
                en<='1';
                a_clr<='0';
                cnt_en_ar<='1';
                cnt_reset<='0';
                cnt_en<='1';
            WHEN disable => --disable en
                en<='0';
                a_clr<='0';
                cnt_en_ar<='0';
                cnt_reset<='0';
                cnt_en<='0';
        END CASE;
  END PROCESS;


END Behavior;
```

8.6. Main: Ring Oscillator PUF

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

---delay_clk: The amount of clock cycles to allow enable to be on.
---ro_PUF_width:The amount of PUFs to have in parallel
---RO_sel: The width of the selector for PUF mux and decoder

---clock:Clock for the state machine
---w:Allows the state machine to leave the start state on high.
---reset: resets the state machine back to start state
---selA: Select an RO
---selB: Select another RO for comparison to the first
---res_ar: The result of each PUF.
entity RO_PUF_Main is
  generic ( delay_clk : integer := 125000;
                ro_PUF_width : integer := 8;
                RO_sel : integer  := 2);
    Port ( clock, w, reset : in  STD_LOGIC;
           selA, selB : in  STD_LOGIC_VECTOR (RO_sel-1 downto 0);
           res_ar : out  STD_LOGIC_VECTOR (ro_PUF_width-1 downto 0));
end RO_PUF_Main;

architecture Behavioral of RO_PUF_Main is

component PUF_Machine IS
  generic (delay : integer := 125000);
  PORT ( clock,w,Reset : in std_logic;
         en,a_clr,cnt_en_ar : out std_logic);
END component;

component PUF_Array is
  generic (puf_Width : integer := 8;
           sel_width_ar : integer := 8);
```

```
  port( en_ar, aclr_ar, cnt_en_ar: in std_logic;
        selA : IN STD_LOGIC_VECTOR(sel_width_ar - 1 DOWNTO 0);
        selB : IN STD_LOGIC_VECTOR(sel_width_ar - 1 DOWNTO 0);
        res_ar: out std_logic_vector(puf_Width - 1 downto 0));
end component;

signal m_en, m_clr, m_cnt_en : std_logic;

begin

state_mac :PUF_Machine GENERIC MAP(delay => delay_clk)
            PORT MAP(clock => clock,
                     w => w,
                     Reset => reset,
                     en=>m_en,
                     a_clr=>m_clr,
                     cnt_en_ar=>m_cnt_en
                     );


puf_ar :PUF_Array GENERIC MAP(puf_width => ro_PUF_width,
                                           sel_width_ar=>RO_sel)
            PORT MAP(en_ar=>m_en,
                     aclr_ar=>m_clr,
                     cnt_en_ar=>m_cnt_en,
                     selA=>selA,
                     selB=>selB,
                     res_ar=>res_ar
                     );


end Behavioral;
```

9. REFERENCES

[1] T. Olzak, “Keystroke Logging (Keylogging),” Apr. 2008.

[2] “HARDWARE KEYLOGGERS.” Centre for the Protection of National Infrastructure, May-2009.

[3] N. Grebennikov, “Keyloggers: How they work and how to detect them (Part 1),” *securelist.com*. [Online]. Available: http://www.securelist.com/en/analysis/204791931/Keyloggers_How_they_work_and_how_to_detect_them_Part_1. [Accessed: 02-May-2013].

[4] Q. Wang, “KeyScrambler.” [Online]. Available: http://www.qfxsoftware.com/ks-windows/how-it-works.htm.

[5] M. Kassner, “How antivirus software works: Is it worth it? | TechRepublic,” *IT Security*, 19-Jan-2010. .

[6] C. Nachenberg, “Understanding and Managing Polymorphic Viruses.”

[7] L. Duy, C. Yue, T. Smart, and H. Wang, “Detecting Kernel Level Keyloggers Through Dynamic Taint Analysis,” College of William & Mary, May 2008.

[8] D. Marcus and T. Sawicki, “The New Reality of Stealth Crimeware,” 2011.

[9] C. Morris, “Playable version of Half-Life 2 stolen,” *CNN*, 07-Oct-2003. [Online]. Available: http://money.cnn.com/2003/10/07/commentary/game_over/column_gaming/.

[10] K. Paulson, “Guilty Plea in Kinko’s Keystroke Caper,” *SecurityFocus*, 18-Jul-2003. [Online]. Available: http://www.securityfocus.com/news/6447.

[11] J. Leyden, “Hardware keyloggers found in Manchester library PCs,” *The Register*, 15-Feb-2011. [Online]. Available: http://www.theregister.co.uk/2011/02/15/hardware_keyloggers_manchester_libraries/.

[12] V. Woolaston, “Computer virus found on Facebook steals bank details and money from accounts when users click on links,” *Mail Online*, 05-Jun-2013. [Online]. Available: http://www.dailymail.co.uk/sciencetech/article-2336388/Computer-virus-Facebook-steals-bank-details-money-accounts-users-click-links.html. [Accessed: 06-Jun-2013].

[13] D. Tom, “Zeus Trojan returns: Facebook being used to spread the infection,” *TechSpot*, 05-Jun-2013. [Online]. Available: http://www.techspot.com/news/52795-zeus-trojan-returns-facebook-being-used-to-spread-the-infection.html. [Accessed: 06-Jun-2013].

[14] A. Maiti and P. Schaumont, “Improved Ring Oscillator PUF: An FPGA-friendly Secure Primitive,” Virginia, 2009.

[15] G. E. Suh and S. Devadas, “Physical Unclonable Functions for Device Authentication and Secret Key Generation,” San Diego, California, 2007.

[16] D. Merli, F. Stumpf, and E. Claudia, “Improving the Quality of Ring Oscillator PUFs on FPGAs,” Scottsdale, Arizona, 2010.

[17] A. Diaz-Perez, N. Saquib, and F. Rodriguez-Henriquez, “Some Guidelines for Implementing Symmetric-Key Cryptosystems on Recon¯gurable-Hardware,” Instituto Politecnico Nacional, Mexico.

[18] F. Mihailowitsch, “Detecting Hardware Keyloggers,” 28-Oct-2010.

[19] A. Davis, “Hardware Keylogger Detection.”

[20] S. Morozov, A. Maity, and P. Schaumont, “A Comparative Analysis of Delay Based PUF Implementations on FPGA,” Virginia Polytechnic Institute and State

University.
[21] C. Costea, F. Bernard, V. Fischer, and R. Fouquet, “Analysis and Enhancement of Ring Oscillators Based Physical Unclonable Functions in FPGAs,” Universite de Lyon.
[22] L. Felton, A. Spilla, M. Sauer, T. Schubert, and B. Becker, “Analysis of Ring Oscillator PUFs on 60nm FPGAs,” presented at the European Cooperation in Science and Technology, Albert-Ludwigs-University of Freiburg.