2018-03-01

A Flexible FPGA-Assisted Framework for Remote Attestation of Internet Connected Embedded Devices

Jared Russell Patten
Brigham Young University

A Flexible FPGA-Assisted Framework for Remote Attestation

# of Internet Connected Embedded Devices

Jared Russell Patten

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering

Brigham Young University

ABSTRACT

A Flexible FPGA-Assisted Framework for Remote Attestation
of Internet Connected Embedded Devices

Jared Russell Patten
Department of Electrical and Computer Engineering, BYU
Master of Science

Embedded devices permeate our every day lives. They exist in our vehicles, traffic lights, medical equipment, and infrastructure controls. In many cases, improper functionality of these devices can present a physical danger to their users, data or financial loss, etc. Improper functionality can be a result of software or hardware bugs, but now more than ever, is often the result of malicious compromise and tampering, or as it is known colloquially "hacking". We are beginning to witness a proliferation of cyber-crime, and as more devices are built with internet connectivity (in the so called "Internet of Things"), security should be of the utmost concern. Embedded devices have begun to seamlessly merge with our daily existence. Therefore the need for security grows as it more directly affects the safety of our data, property, and even physical health.

This thesis presents an FPGA-assisted framework for remote attestation, a security service that allows a remote device to prove to a verifying entity that it can be trusted. In other words, it presents a protocol by which a device (be it an insulin pump, vehicle, etc.) can prove to a user (or other entity) that it can be trusted - i.e. that it has not been "hacked". This is accomplished through executable code integrity verification and run-time monitoring. In essence, the protocol verifies that a device is running authorized and untampered software and makes it known to a verifier in a trusted fashion. We implement the protocol on a physical device to demonstrate its feasibility and to examine its performance impact.

Keywords: Remote Attestation, Embedded Systems Security, Reconfigurable Computing, FPGA, Internet of Things, IoT

ACKNOWLEDGMENTS

First and foremost I would like to thank my amazing wife. I'm grateful for her patience and willingness to put up with my late night study sessions, and for the kind words of encouragement through the process.

I would like to thank my thesis advisor Prof. Archibald, who has continually inspired and motivated me in my studies. His knowledge, enthusiasm, and patience have been invaluable to my research. Additionally, I would like to extend my thanks to my thesis committee: Prof. Wilde and Prof. Lee for their mentoring and valuable feedback.

I extend special thanks to all my professors and colleagues at BYU who have contributed to my education in so many ways. I'd also like to thank my fellow members of the BYU Embedded Security Group for their contributions, ideas, feedback, and for making it a fun experience.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

CHAPTER 1. INTRODUCTION

In 2015, a team of security researchers demonstrated that they could remotely gain control of a 2015 Jeep Cherokee [1]. Through an insecure remote update process, the team was able to modify the firmware of a specific microcontroller on the vehicle, which allowed them to issue arbitrary commands over the vehicle's CAN bus [2]. With full control over the vehicle's CAN bus, the attackers could assume control of the vehicle's lights, wipers, brakes, and transmission.

Like modern vehicles, many systems feature external connectivity through Internet enabled embedded devices. These devices increasingly permeate our everyday lives; we find them in our vehicles, traffic lights, medical equipment, power grid controls, etc. There are currently an estimated 6.3 billion connected devices in the so called "Internet of Things" (IoT), and that number could reach 20.4 billion by 2020 [3]. Improper functionality of these devices can present a physical danger to their users, data or financial loss, etc [4], [5]. Improper functionality can be the result of software or hardware bugs, but now more than ever, it can also be the result of malicious software. We are beginning to see a proliferation of cyber-crime, therefore the need for secure embedded devices is greater than ever. The problem was magnified in 2016 by the massive distributed denial of service (DDoS) attack against DNS servers, making Twitter, Netflix, Reddit, and other websites inaccesible for several hours [6]. The attack was carried out by compromised IoT devices, primarily web cameras, DVRs, and routers infected with the Mirai Botnet.

There are many common security vulnerabilities that plague embedded systems, such as weak cryptography, software vulnerabilities, and unchanged default passwords [7], [8]. Buffer overflows are one of the most common and well-known types of software vulnerabilities [9]. Under the right circumstances, a buffer overflow exploit can allow an attacker to insert malicious code into device memory [10]. Buffer overflows are only one example of a vulnerability through which an attacker can insert malicious code into a system.

Most current embedded systems have no means of detecting malicious code in their memory. Malicious code can thus go unnoticed until it has released its payload or inflicted damage on the system, as was the case with the 2010 Stuxnet worm that slowly sabotaged and destroyed Iranian nuclear centrifuges over a period of several months [11].

While it is difficult, if not impossible, to design a perfectly secure system, we can design systems that are capable of detecting tampering. An active area of research for tamper detection is a method called *remote attestation*. Francillon et al. define remote attestation as: “A security service that involves verification of internal state of a remote embedded device” [12]. Many proposed remote attestation solutions work by comparing the remote device memory contents with a “golden” model of memory [13]–[17]. Usually, the memory contents are converted to a fixed-length representation with a hashing or checksum algorithm (described in more detail in Section 2.3.1) to simplify transmission and storage of the golden model. A device whose code has been modified will produce an incorrect checksum, and a remote verifier can then choose how to respond, perhaps by halting communication with the device or notifying an administrator or user. Many proposed protocols generate this checksum purely in software [13], [15]; others require custom hardware architectures or ASICs [16], [18], [19].

Remote attestation protocols which rely on memory integrity verification techniques face several major challenges. First, as embedded systems grow in sophistication, their memory map also grows in complexity and size. Therefore, taking a hash or checksum of an entire memory map can hog computing resources. Software-only solutions pose a challenge because the checksum-generating code itself could become compromised. Generally, hardware solutions are more secure, but are more expensive and inflexible.

The contribution of this thesis is a flexible FPGA-based remote attestation framework that builds primarily upon two previously proposed solutions. We observe that the memories of modern embedded systems contain not only executable code, but also large regions of runtime heap, stack memory and memory mapped IO (i.e., dynamic regions of memory). To predictively generate a golden checksum of these dynamic memory regions would be difficult, if not impossible. Instead of taking a checksum of the entire memory layout, the FPGA-based security service proposed in this thesis periodically takes a checksum of the software executable, directly followed by an assertion of the memory address of the currently executing instruction. If this address lies outside of a pre-defined legal range, or the checksum does not match the golden checksum, the FPGA-based attestation service internally records a device compromise. Through encrypted communication, a remote trusted device can query the security status of the untrusted device by communicating with the FPGA security service.

We implement and demonstrate the feasibility of this framework and observe how it affects system performance. Additionally, we analyze and discuss its security properties. We show how the framework can be adapted for different types of embedded systems, therefore providing a flexible, secure remote attestation framework for embedded systems.

The remainder of this thesis is organized as follows. Chapter 2 presents a definition of remote attestation and defines a set of required security properties. Chapter 3 is a brief survey of previously proposed remote attestation solutions. Chapter 4 defines a threat model, which we will use as a basis for our security analysis in a later chapter. In Chapter 5 we present our proposed attestation framework and implementation details. Chapter 6 consists of the results and analysis of our implementation. Lastly, Chapter 7 discusses our conclusions and future work.

CHAPTER 2. REMOTE ATTESTATION

2.1 Definition

Remote attestation is the process of remotely (usually over an Internet connection) verifying the internal state of a system. This state can include code memory, hardware configuration, runtime state, etc. Vulnerabilities (as described in Chapter 4) in embedded systems may allow a malicious entity to assume control of a device or extract sensitive information. Generally, this requires an attacker to modify the system in some way (i.e. through code insertion) so that the system can be re-purposed to suit the attackers needs. The goal of remote attestation is to detect malicious tampering remotely. It is important to understand that remote attestation techniques do not attempt to prevent tampering, but instead aim to detect it.

Remote attestation is generally implemented as a challenge-response protocol between a *verifier* entity and a *prover* entity. In the computer security field, a challenge-response protocol is where one entity prompts (challenges) another entity to provide an answer (response). The challenger then decides whether the answer is correct, and acts accordingly. Password authentication is a common example of a challenge-response protocol. In the context of remote attestation, we define the challenger as the *verifier*, and the responder as the *prover* (see Figure 2.1). Remote attestation protocols are not limited to one prover for each verifier, in fact a single entity may act as the verifier to many provers.

When challenged by the verifier, a prover is responsible for constructing a *proof*, which usually consists of a representation of its internal state. We assume that the verifier is trusted, while the prover is untrusted. Therefore, the proof must be unforgable, so that even a compromised device is incapable of providing false statements about its state. When challenged by a verifier, the prover sends a proof in response, which the verifier uses to establish trust in the device.

This uncovers the major paradox and challenge with remote attestation. How can we guarantee that a proof from an untrusted device has not been forged? Chapter 3 presents a brief survey

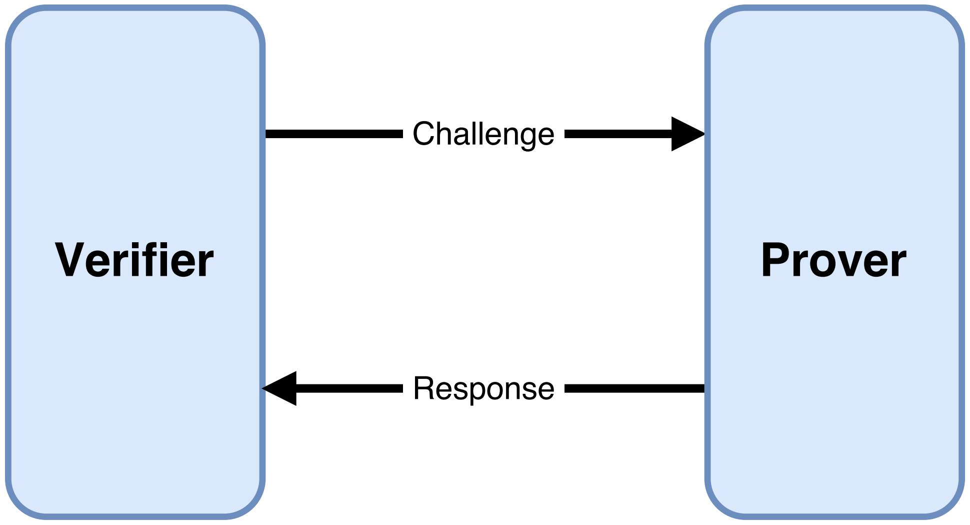


Figure 2.1: Remote attestation is challenge-response protocol between verifier and prover entities. The verifier initiates a “challenge”, to which the prover replies with a “response”.

of several proposed implementations of remote attestation, each of which attempt to solve this problem.

2.2 Example Use Case

As of this writing, almost half of all American households are equipped with “smart” power meters [20]. As opposed to traditional meters, smart meters are characterized by the capability of two-way communication between the meter and the utility, usually via Wi-Fi, cellular, or wired connection (see Figure 2.2) [21]. Smart meters allow for real-time sensor readings, which benefit the utility through improved capability for power outage notification, increased power reliability, and detailed monitoring. Consumers benefit by gaining real-time energy usage information, allowing them to better manage energy use, in turn lowering energy costs.

While smart meters have the potential to increase the efficiency of energy distribution and consumption, they also pose a greater security challenge than traditional meters. Smart meters in Puerto Rico have already been successfully exploited, as was revealed in a 2010 FBI intelligence bulletin [22]. According to the assessment, the compromised meters were programmed to under-report electricity usage, resulting in estimated annual losses of $400 million.

These particular meters feature an optical port, through which a field technician can gather in-field diagnostics. Using an optical converter, similar to the one used by technicians, an attacker

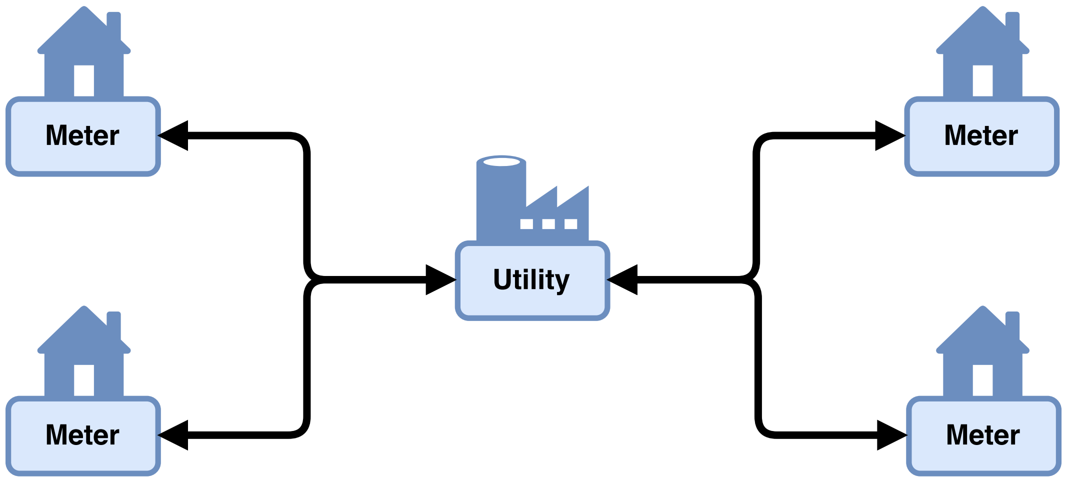


Figure 2.2: Smart meters are characterized by two-way communication between the utility and the meter.

can communicate with the meter using a computer and special software. At this point, the attacker has full control over the device configuration, and can configure it to under-report, reducing the customer bill by 50 to 75%.

The FBI believes that former employees of the power utility were responsible for carrying out the attacks, as well as for training others to do so. Hackers were known to charge between \$300-\$1000 dollars to alter a meter, resulting in a cheaper power bill to the customer. Thus, there was monetary incentive for both the hackers and the customers.

Although physical access was required to carry out this particular attack, it is not far-fetched to imagine a future where Internet connected smart meters are *remotely* vulnerable to similar types of tampering. Without remote attestation (or a similar protocol), how can the utility possibly ensure that a meter has not been infected with malware or been compromised in some manner?

Effective remote attestation could play a crucial role in assuring that smart meters have not been tampered with, and therefore are reporting accurate information. Timely detection of compromised meters could have massive financial benefits to the utility and may even serve as a deterrent to hackers; if an attacker knows there is a high probability of detection, there is likely less motivation to attempt the hack.

Under our definition of remote attestation, each smart meter corresponds to a prover entity, and the power utility is the verifier entity. When challenged by the utility, the smart meter must provide a proof of its internal state, thus proving to the utility that it has not been compromised.

Perhaps the utility wishes to attest each smart meter at periodic intervals, or possibly just right before receiving a meter reading so as to ensure that the readings are authentic.

As with any security related topic, it is difficult to quantify what makes a system "secure", as security is largely a cat and mouse problem. However, in the following sections, we explore several properties required to defend against currently known attacks.

2.3 Required Security Properties

A system designer cannot fully anticipate the vulnerabilities that may appear in a system, but certain classifications of vulnerabilities have been well studied and can be prevented [23], [24]. Based on the works of Coker et al. and Francillon et al., we define a minimal set of properties that are required for a secure remote attestation protocol [12], [25], which we use as a basis in the design of our proposed attestation framework. The six properties of the framework are as follows:

1. Measurement Diversity
2. Domain Separation
3. Self-Protection
4. Exclusive Key Access
5. Immutability
6. Controlled Invocation

Most implementations of remote attestation strive to meet these requirements through various means. Following is a more thorough discussion of each of the these properties.

2.3.1 Measurement Diversity

When challenged by a verifier, the prover responds with a proof representing its internal state. The state of a device may consist of the loaded executable, memory state, hardware configuration, etc. Thus, a prover device must be capable of taking self-measurements. The claims of a proof are limited by the types of self-measurements taken by the prover. For example, if a proof

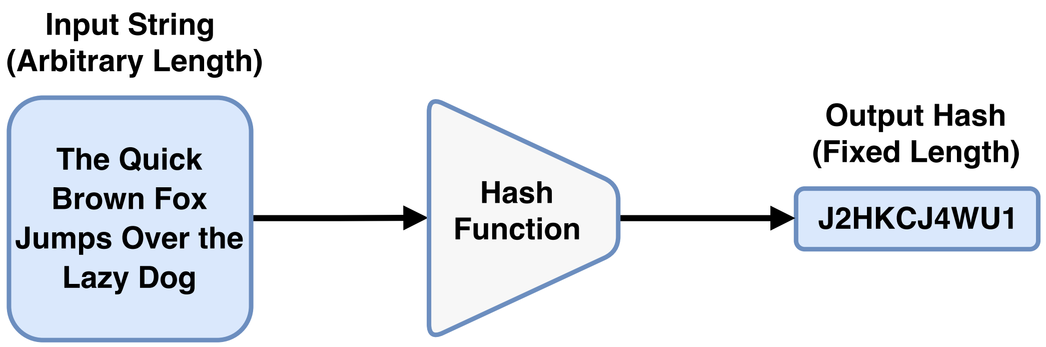


Figure 2.3: A hash function takes a string of finite length input and produces a fixed-length output. In this example, the input string is mapped to a 10 character alpha-numeric string. Cryptographic hashes like this are often used in remote attestation protocols.

consists solely of memory contents, then the prover can only make integrity claims regarding its memory; the proof says nothing about the hardware configuration. Indeed, the verifier can only attest the prover memory.

To construct a memory proof, many attestation protocols compute a cryptographic hash of memory. A hash function is simply a one-way function that takes an arbitrary length input, and produces a fixed-length "hash" (see Figure 2.3). Hash functions are many-to-one, meaning that many different inputs can produce the same hash. Cryptographic hash functions are a sub-class of hash functions that ideally exhibit the following three properties [26]:

1. Preimage resistance - Given a hash function h and output y, it is computationally difficult to find an input x such that $h(x) = y$.

2. 2nd-preimage resistance - Given a hash function h and an input x, it is computationally difficult to find an additional input x' such that $x \neq x'$ and $h(x) = h(x')$.

3. Collision resistance - Given a hash function h, it is computationally difficult to find two values x, and x' such that $x \neq x'$ and $h(x) = h(x')$. In other words, it is difficult to find two input values that result in the same hash.

One can see why cryptographic hash functions are useful for remote attestation; they allow attestation code or hardware to compute a small fixed length value, which can then be supplied to a verifier instead of sending the entire memory contents.

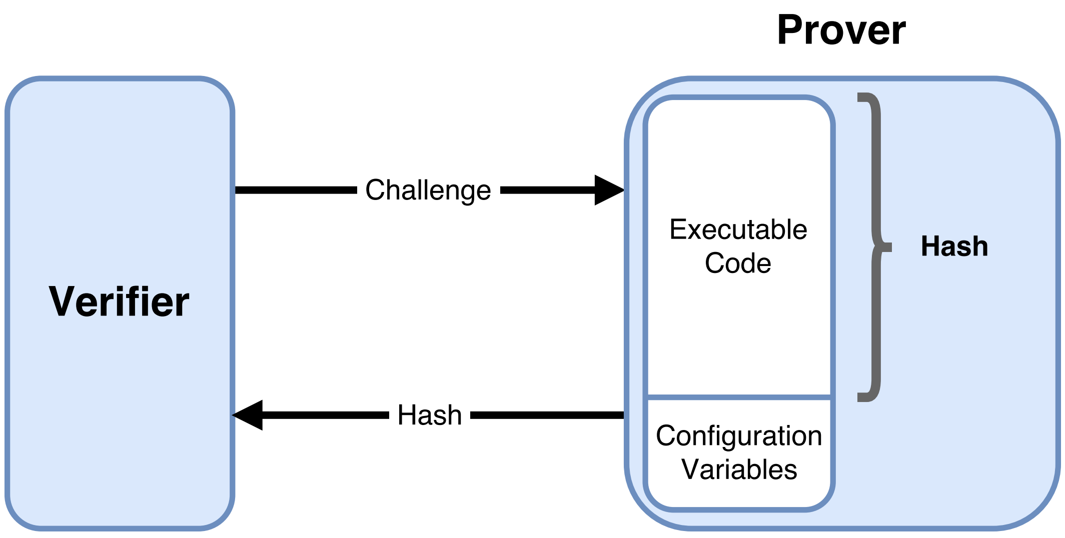


Figure 2.4: Secure remote attestation protocols should perform diverse internal state measurements. This Figure illustrates a remote attestation scheme under which only the executable code is attested, resulting in a non-diverse measurement.

Recall from Section 2.2, that the configuration of Puerto Rican smart meters was simply modified to under-report power usage. Let us imagine that the meters were configured to under-report by simply modifying a system variable, implying that the executable code did not require modification. Imagine that the meter has a built-in remote attestation protocol that when challenged, computes a cryptographic hash of the executable code, then returns the hash to the verifier. The verifier can compare this hash against its golden hash, and respond accordingly (see Figure 2.4).

Note that the modification of system variables will not affect the resulting hash during an attestation sequence. Therefore, modification of system configuration variables does not manifest itself in the proof, resulting in a false positive from the perspective of the verifier.

This example illustrates the importance of taking diverse measurements of the system state. Ideally, a prover should be comprehensive in measuring its internal state. As mentioned earlier, the system state may consist of both hardware and software.

2.3.2 Domain Separation

Domain separation is the idea that different pieces of system functionality can by separated into different *domains*, such that the interface between them is finely specified and tightly controlled. Therefore, two different domains can only interact with one another in a pre-determined fashion.

With remote attestation, it is vital that measurement tools are able to provide accurate and untampered measurements of the application, even when the application is compromised. In this context, measurement tools (or other secure functionality) must reside in a separate domain from the application, such that the application cannot tamper with the measurement tools. Therefore, even if the target is compromised, the prover can still perform accurate and reliable measurements of the system.

Figure 2.5 illustrates an example system in which the measurement tools, a secure boot module, and an Advanced Encryption Standard (AES) engine are placed in a separate domain from the device code and hardware configuration. We define the interface between the domains as one-way from the measurement tools to the device code and hardware configuration. Therefore, even if device code is compromised – i.e. through malware – it cannot tamper with the measurement tools.

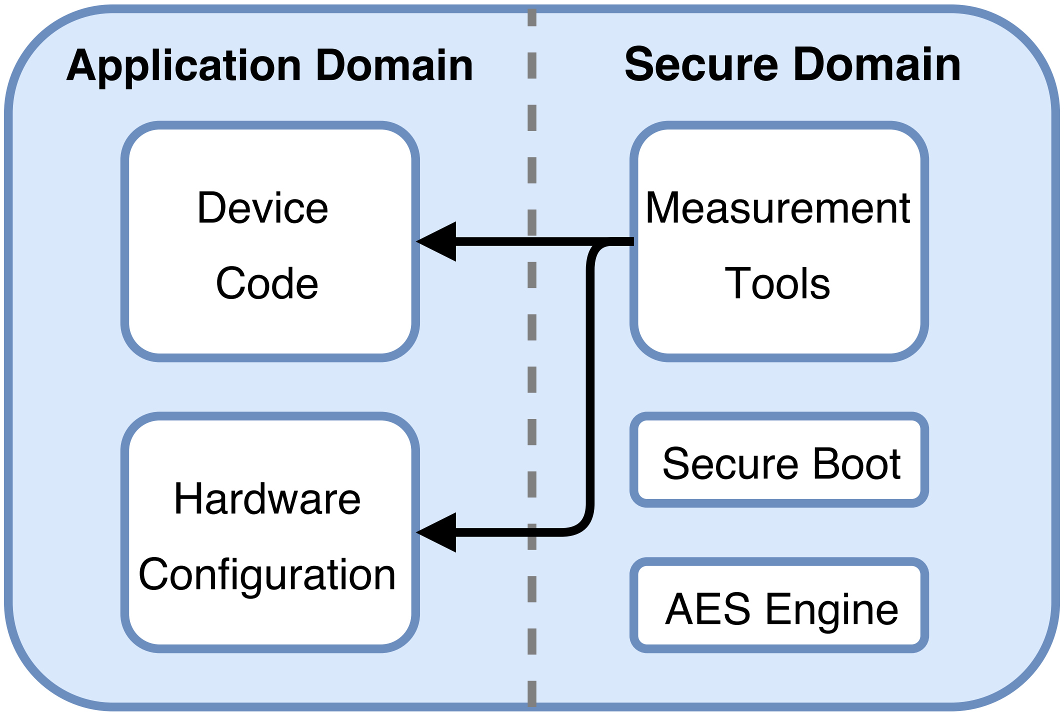


Figure 2.5: Example domain separation.

One common method to achieve domain separation is through virtualization techniques [27]. For example, measurement tools could be placed in one virtual machine (VM), and the application domain in another. The system could be configured so that the measurement tool VM has read access to the target VM, but not the other way around [15]. The software programs that run virtual machines are called *hypervisors*. If we make the assumption that a hypervisor is implemented correctly, a compromised target does not have the ability to tamper with the measurement tools, as they are contained in separate VMs.

The downfall of virtualization techniques are that they often require more processing power, as the computational overhead introduced by a hypervisor is non-trivial. To combat this issue, several light-weight hypervisors have been developed that are optimized for small embedded devices [27]–[30]. Container based virtualization is another promising technology for implementing domain separation, where virtualization takes place at the operating system (OS) level, as opposed to hypervisors which must run a guest OS on top of a host OS [31]. Containers don't require special hardware like some hypervisors do, and often require less processor overhead [32], [33].

Hardware assisted techniques are arguably more secure due to the nature of physical separation, but can be more costly. Copilot [34] and trusted platform modules (TPM) [35] are two examples of techniques for achieving domain separation through hardware.

2.3.3 Self-Protection

Self-protection refers to the problem of building trust in the integrity of the domain separation itself. Most attestation frameworks consist of a trusted and a non-trusted domain, but what properties of the trusted domain allow us to trust it? How do we initially build trust in the secure domain? These are the fundamental problems that self-protection presents.

To trust a domain means that we do not need to attest it. In other words, we can trust the domain even if remote attestation is not performed on it. Sometimes, we build trust through formal verification of the domain. Formal verification is the act of verifying or proving the correctness of a design, either hardware or software. Because of the large number of input combinations, as well as internal state, large systems are extremely difficult to formally verify. Thus, for a domain to be trusted via formal verification, it must be built with components small enough that they can be formally verified.

Formal verification is a viable option because vulnerabilities in a system are often the result of software or hardware bugs. Therefore a formally verified hardware or software system is difficult to exploit.

ARM processors contain secure-boot, another method for establishing trust. The general idea is that you start with a small trusted component (possibly trusted through formal verification) called the root of trust. Generally the root of trust must reside in a non-modifiable section of code, such as read-only memory (ROM), or in hardware. A chain of trust is then established by authenticating every other component before it is used. A TPM does this by having a hardware root of trust, which then cryptographically authenticates the integrity of any software before it is used by the system processor.

2.3.4 Exclusive Key Access

Many remote attestation frameworks rely on some form of cryptographic communication or other cryptographic operations. If this is the case, the attestation components must have their own keys that are inaccessible to other components of the system. Other system components must not have access to these keys, so as to prevent any type of forgery or abuse by a compromised system. For example, keys could be stored in the secure domain in such a way that they are physically inaccessible from the application domain.

Imagine an attestation protocol where a prover takes a cryptographic hash of its executable and sends it to the verifier. The issue here is that if a compromised target discovers the hashing algorithm used by the attestation protocol, it becomes easy for the adversary to forge a valid memory hash by storing a "good" copy of the executable somewhere else in memory. Figure 2.6 shows what a compromised memory map may look like in such a scenario.

To overcome this problem, many protocols utilize a hash-based message authentication code (HMAC). To construct an HMAC, a secret cryptographic key is hashed together with the data to produce a hash that is not reproducible by an entity that doesn't know the secret key. The definition of an HMAC as described by RFC 2104 [36] is as follows:

$$HMAC(K, text) = H((K \oplus opad)||H((K \oplus ipad)||text)), \tag{2.1}$$

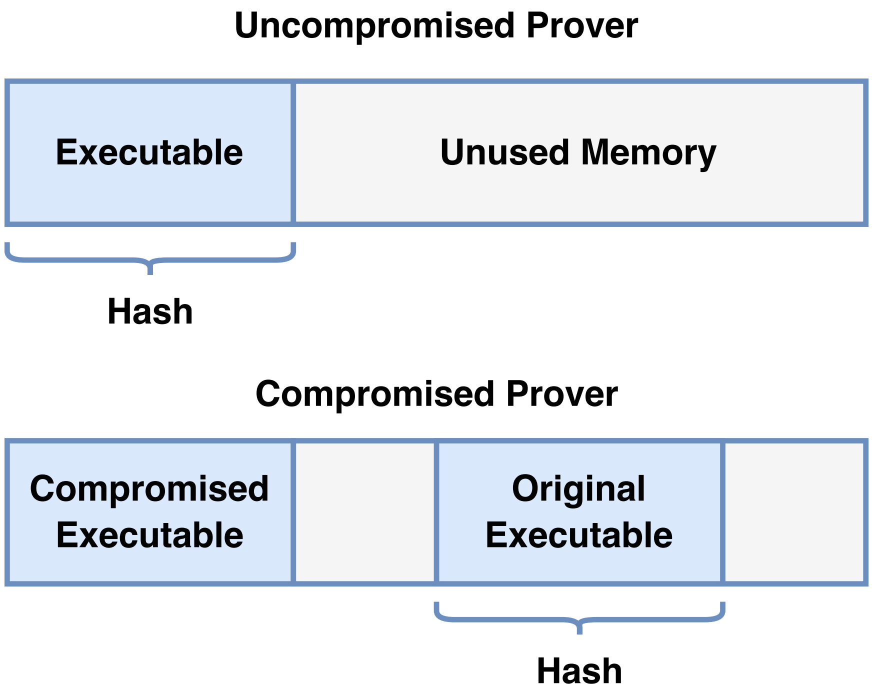


Figure 2.6: Example compromised attestation code. A compromised application may be able to store a copy of the original attestation code in an unused portion of memory.

where:

1. *H* is a cryptographic hash function,
2. *opad* is the byte 0x5C repeated *B* times, where B is the block length (see RFC 2104),
3. *K* is a cryptographic secret key,
4. *ipad* is the byte 0x36 repeated *B* times, where B is the block length (see RFC 2104),
5. *text* is the input data to the HMAC function, and
6. || represents concatenation.

Now, if the attestation components of the system have exclusive access to *K*, then even a compromised executable cannot forge a hash of the original executable. This is only one example (albeit a common one) of secret key usage in remote attestation schemes. Cryptographic or authenticated communication between the prover and the verifier is another common use.

2.3.5 Immutability

Immutability follows closely with domain separation, but it is simply the notion that attestation code or hardware should not be modifiable by other components of a system. Hardware attestation is inherently immutable, because it is impossible for a compromised system to modify hardware functionality. Sometimes FPGAs are used for attestation functionality and although it is possible to modify an FPGA bitstream from software, certain techniques can be employed to make this difficult. A brief discussion on FPGA bitstream security is given in Section 5.2.1. If the attestation functionality resides in software, then a mechanism to prevent modification of attestation code is necessary. This can be accomplished with read-only-memory (ROM), or some other such protection. Domain separation is also a good technique to prevent unauthorized components from accessing attestation code, therefore preventing modification.

2.3.6 Controlled Invocation

Only an authorized verifier should be allowed to initiate attestation. This property ensures that an adversary cannot initiate attestation, which is important in preventing denial-of-service attacks. Denial-of-service attacks, as the name describes, are a class of attacks where an adversary attempts to prevent access to some type of resource. Denial-of-service attacks are discussed in more detail in Section 4.2.

CHAPTER 3. RELATED WORK

This chapter presents a brief survey of relevant research in the area of remote attestation. We focus on several techniques that have influenced the work presented in this thesis. Of particular importance to this work are SWATT and FPGA-based memory verification.

3.1 Software Remote Attestation

Remote attestation techniques that do not rely on external hardware or custom architectural solutions are desirable in many situations, such as for legacy systems where hardware modifications are impractical. While desirable for certain types of applications, they tend to be inherently less secure, as software is generally more exploitable than hardware. What follows is a discussion of several prominent software attestation techniques.

3.1.1 XEn Based Remote Attestation (XEBRA)

As discussed in Chapter 2, a secure attestation protocol should contain strongly defined domain separation and have the ability to self-protect its secure functionality. XEn Based Remote Attestation (XEBRA) [15] implements a form of domain separation by utilizing the lightweight hypervisor Xen. The goal is that two separate virtual machines will not be able to interfere with one another. The Xen hypervisor is chosen because it is small enough in size to be formally verified, thus giving reasonable assurance of the correctness of the hypervisor.

The XEBRA framework consists of two separate virtual machines running on the same hardware system, referred to as the control and application domains, respectively. Through secure boot technology, the control domain acts as the root of trust for secure functionality. The attestation functionality is implemented in the control domain, and therefore boots into a trusted state. Thus, if application code is compromised, an attacker will not be able to modify attestation or other secure functionality because it resides in a separate virtual machine.

Obviously, XEBRA is only well suited to devices that provide hardware virtualization support, and therefore is not suitable for resource constrained embedded devices. The XEBRA paper argues that because the cost of microcontrollers is continually declining, and more embedded processors (such as many ARM family devices) are beginning to include hardware virtualization support, the feasibility of the XEBRA protocol will continue to grow.

3.1.2 SoftWAre-based remote ATTestation (SWATT)

In SoftWAre-based remote ATTestation (SWATT) [13], no custom hardware is required. The entire protocol can be implemented purely in software. The protocol goes as follows:

1. Verifier initiates the attestation by sending a challenge to the prover. The challenge contains a pseudo-random value.
2. Prover uses the value received from step 1 to seed a pseudo-random number generator (PRNG).
3. Prover uses the PRNG output to generate a memory address.
4. Prover performs a memory read at the address from step 3, and uses the result to update a checksum.
5. Prover repeats steps 3 and 4 for a predetermined number of iterations (using the next PRNG value at each step).
6. Prover sends the resulting checksum to the verifier.
7. Verifier compares the checksum against a "golden" checksum.
8. If the prover response time is longer than a pre-determined value, the verifier assumes that the checksum-generating memory traversal has been tampered with and therefore deems the prover to be compromised.
9. If the checksums match, the attestation is successful, otherwise it fails and the device is assumed to be compromised.

The Pseudo-random traversal of memory is designed to prevent a compromised prover from simply pre-calculating the correct checksum and returning this to the verifier. This way, the prover cannot predict the checksum and can only begin calculating it when the attestation request is received.

A compromised device could potentially store the original executable code in some unused portion of memory, then redirect all memory loads of the pseudo-random traversal to the original executable (see Figure 2.6). For example, imagine we have an executable of 1000 bytes starting at address *0x0000*. We assume that all memory locations beyond address *0x03E8* (1000 in hexadecimal notation) are unused. The compromised device makes a copy of the original executable and stores it in memory starting at address *0x03E8*. Therefore, if the first address of the pseudo-random traversal is address *0x0005*, then the load will can be redirected to address *0x03ED* (1005 in hexadecimal) where the original contents of address *0x0005* reside. In this manner, the compromised device can generate a correct checksum.

To prevent this scenario, the response time of the prover is taken into consideration. Load redirects would require an additional if-statement in the code to check if the next memory read touches an altered location. Therefore, the additional time required to execute the if-statement would be manifest in the prover response time. This is only possible, however, if the network latency is deterministic.

A major advantage of SWATT is that it makes no assumptions about the integrity of the checksum-generating function. In other words, attestation should function properly even if the application (including the checksum-generating function) is compromised.

3.2 Hardware Remote Attestation

As seen in Section III, software-only attestation techniques suffer from several challenges, such as self protection of attestation functions. Generally, techniques that leverage external hardware can help to alleviate some of these challenges. What follows are two prominent attestation techniques involving secure coprocessors or hardware modifications.

3.2.1 SMART

Secure and Minimal Architecture for (establishing a dynamic) Root of Trust (SMART) [18] uses a custom architectural solution to provide a secure root of trust. This root of trust can be used to create a secure attestation scheme.

SMART works by storing attestation code on a prover device in an immutable read-only memory (ROM). This code basically generates a checksum of a memory region as specified by a verifier device. Because attestation code is stored in ROM, a compromised application will not be able to modify it.

The checksum is a keyed HMAC, consisting of code memory. The HMAC key is private, and therefore is only accessible by SMART attestation code. To prevent unauthorized key access, SMART utilizes a custom memory controller unit (MCU). The MCU protects access to the key by only allowing access if the contents of the PC register currently points to code in the ROM SMART region. In other words, only SMART code is allowed to read the key in memory. An attempted access to the key with an invalid PC value will cause the MCU to reset the processor, thus thwarting unauthorized key access. To prevent any abuse of SMART code, the processor is only allowed to execute the ROM SMART code at a pre-defined starting location, which is enforced by the MCU in the same fashion as the key access protection mechanism.

SMART aims to provide strong security properties, while requiring minimal hardware modifications. The authors claim that many devices already have built-in ROMs and that the MCU modifications are few in number and relatively easily accomplished. The SMART paper demonstrates the feasibility of the protocol by describing a hardware implementation. This was accomplished by modifying an open source MSP430 core in the SystemVerilog hardware description language, doing so with less than 200 lines of modifications to the code.

3.2.2 FPGA-Based

Basile et al. present a remote code verification protocol based on a hybrid FPGA-processor approach [17]. An FPGA is used as a root of trust for secure functionality, and a processor is used for user application functionality, called the secure and application domains respectively. Both the FPGA and processor work in tandem as a prover device. Communication between the verifier and the prover device is handled by the processor, which forwards attestation requests to the hardware secure domain. An attestation sequence takes the following steps:

1. Verifier initiates the attestation by sending an encrypted request. Both the verifier and the secure (FPGA) domain of the prover share a symmetric key.

2. Application domain of the prover receives the encrypted request and forwards it to the secure domain.

3. Secure domain decrypts the request and verifies its authenticity.

4. If the request is authentic, the secure domain takes a checksum of: 1) the operating system kernel, and 2) the portion of the program currently in memory.

5. The two checksums from the previous step are combined into an HMAC response, encrypted and sent to the application domain.

6. Application domain forwards the response to the verifier.

7. Verifier determines if HMAC is valid, and responds accordingly.

Note that in steps 2 and 6, if a compromised application fails to forward a response or request, the verifier will assume system compromise. This is possible because in the scenario that a verifier never receives a response from the prover in a pre-defined time frame, the verifier can assume that the prover has been compromised (or had a system failure). Given a sufficiently strong key and symmetric encryption algorithm, a compromised prover will not have the ability to tamper with or extract any sensitive information from the encrypted response or request packets.

CHAPTER 4. THREAT MODEL

In this section we define a threat model, consisting of a summary of several vulnerabilities commonly found in embedded systems, as well as common attacks. In a later section, we analyze the security of our proposed attestation protocol with regards to the threat model. Note that this is not a complete list of threats, only a small subset of very common and well studied ones.

4.1 Software Vulnerabilities

Software bugs are inevitably present in any type of software system. Bugs generally result from human programming or design errors. Sometimes (albeit more rare) bugs can also arise from compiler errors. Regardless of the source, bugs are an inevitable byproduct of software design.

Often, these bugs simply result in incorrect operation of a system, producing unexpected and incorrect results. Sometimes software bugs are simply annoying, but in some cases can have deadly consequences as was the case with the Therac-25 incident [37]. The Therac-25 was a radiation therapy machine, designed for use in cancer treatment, which worked by blasting malignant cells with controlled ionized radiation. Over a period of two years, at least six patients were administered deadly doses of radiation due to a software bug.

Under the right circumstances, software bugs can be exploited by an adversary. Such bugs are referred to as *software vulnerabilities* [8]. By taking advantage of a software vulnerability, an adversary may be able to take control of a system, steal information, cause physical harm or damage, deny access to resources, etc. Although the Therac-25 incident was not the result of malicious intent, similar software bugs could be exploited for malicious purposes.

The well-known 2010 Stuxnet worm was able to propagate and overwrite the code of programmable logic controllers (PLC) that controlled Iranian nuclear centrifuges through several *zero-day* vulnerabilities in Microsoft Windows [38]. A zero-day vulnerability is a type of software vulnerability that is known only to an adversary, and is thus particularly dangerous because it does not usually get fixed until it has been abused for malicious intent.

The software running on embedded systems is no less prone to software vulnerabilities than traditional desktop systems. A 2014 study of embedded firmware security by Costin et al. revealed that a large number of embedded systems are running outdated firmware with software vulnerabilities [39]. Fortunately, there are ways to mitigate software vulnerabilities. Many software vulnerabilities are well documented and can easily be prevented by careful design and implementation.

As previously discussed, formal verification can be used to verify the correctness of a program. Several tools exist to accomplish this task, but as discussed, only small programs are feasibly formally verified. Several programming languages have been developed to help prevent certain types of programming errors. One such language, Rust, helps guarantee memory safety [40]. One type of vulnerability that arises from misuse of memory bugs is discussed in Section 4.3.

4.2 Denial of Service

A denial-of-service attack (DoS) is a category of attack where an adversary attempts to deny legitimate usage of a service to its normal users. Generally, this applies to network services, where the DoS perpetrator attempts to overwhelm the network service, making it unavailable for its intended use. For example, an attacker may attempt to send a high volume of network traffic to a website server in an attempt to overload the server, thus preventing legitimate traffic from accessing it.

A distributed denial of service (DDoS) attack is a type of DoS attack where the flood of traffic comes from many sources. This is usually accomplished through armies of compromised "zombie" devices, called a Botnet (see Figure 4.1) [41]. A zombie is a device that has been infected with some type of Botnet malware such as Mirai. Zombie devices can lie dormant until the Botnet owner wishes to initiate a DDoS attack, at which point the actions of the zombie device are still generally undetected.

DDoS attacks are particularly difficult to thwart because it is difficult to distinguish between legitimate and flood traffic. In a simple DoS attack, the victim can simply block traffic from the

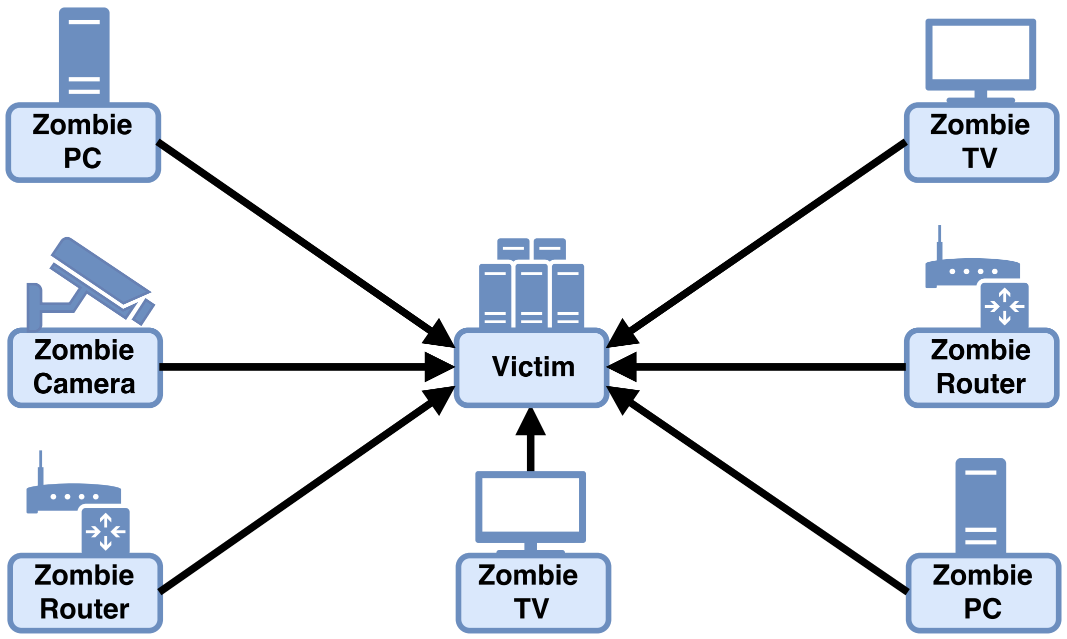


Figure 4.1: IoT Botnets are commonly used for DDoS attacks where many compromised zombie devices are controlled by an attacker. The zombie devices flood a victim with traffic in an attempt to overwhelm it, making its services unavailable to the intended recipients.

attacker's IP address once the attack is detected, whereas DDoS traffic will have many different IP address sources. A zombie device may also purposely only use a small fraction of the host's bandwidth to remain inconspicuous.

The motives behind DDoS attacks vary, but they have been used for activism, blackmail, economic gain, etc. Botnets can essentially be rented, where the price usually depends on the desired duration of DDoS attack. The high availability and ease of access to DDoS attack tools has led to a proliferation of these types of attacks in the wild. Additionally, the dramatic increase in the number of insecure Internet connected devices has only exacerbated the problem.

4.3 Buffer Overflow

Buffer overflows are a common type of software vulnerability that can be particularly dangerous under the right circumstances. This is because they sometimes permit an attacker to insert arbitrary exploit code into a device, sometimes allowing him to take complete control of the device. They are possible because of the way stack frames are laid out in memory (see Figure 4.2 for a sample stack layout).

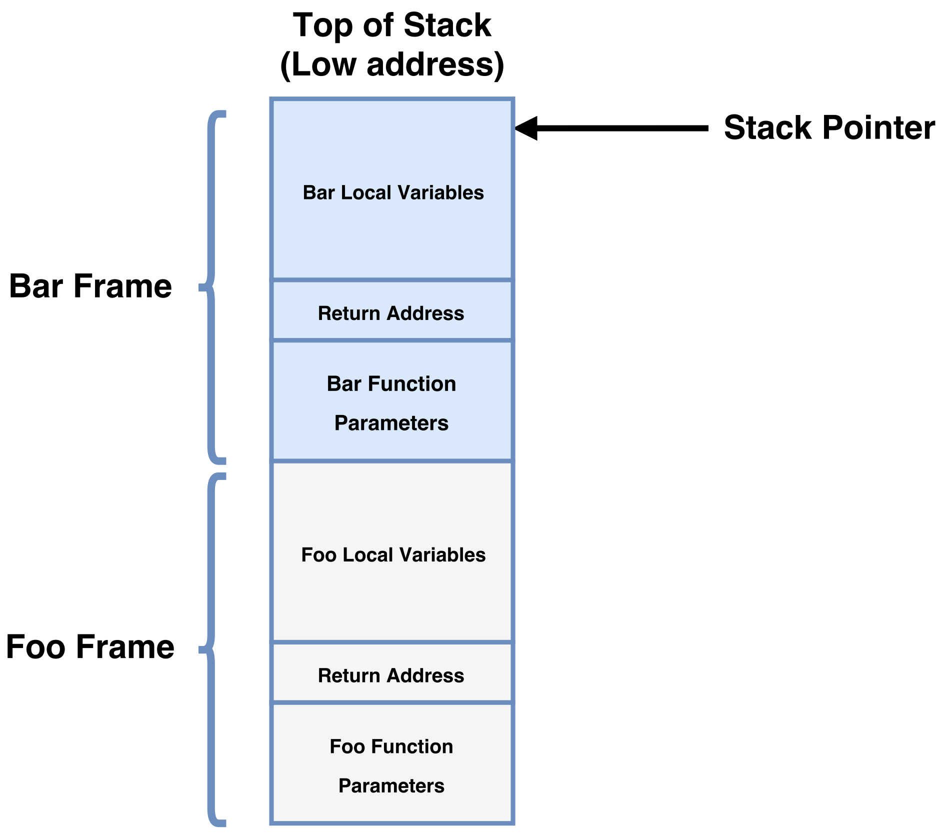


Figure 4.2: Example Stack containing two stack frames, one for function “foo” and another for function “bar”.

Buffer overflow exploits work by writing data to a buffer that is larger than the allocated size of the buffer, thus overwriting adjacent memory. A careful attacker can deliberately overflow a buffer, and overwrite the stack frame return address with a desired target address. Once the function is complete, control will be diverted to the attacker’s desired target address.

A common source of buffer overflow vulnerability is the strcpy function in the C programming language. In it, a source buffer is copied to a destination buffer without regard to size of the destination buffer. For example, the following code segment in listing 4.1 demonstrates a function that is vulnerable to a buffer overflow.

```c
void bar(char* input_buffer)
{
    char buffer[8];
    strcpy(buffer, input_buffer);
}
```

Listing 4.1: Vulnerable C Code Segment

Imagine an attacker who wishes to divert program control to a function located at address *0x12345678*. We assume that the attacker has the ability to provide arbitrary input to the function *bar*, perhaps through system input. Additionally, we assume that the system in question constructs stack frames as in Figure 4.2 and has a 32-bit word size.

To divert control to address *0x12345678*, the attacker must call the function *bar* and provide it with some input (called an exploit string) that will overwrite the return address with the desired target address. The input string to *bar* need only contain 8 bytes of arbitrary data (to fill *buffer*) followed by the desired target address. Figure 4.3 shows how the stack will look after a successful buffer overflow exploit. Notice that the return address has been overwritten with *0x12345678*. Now when *bar* is finished executing, the program will continue execution starting at address *0x12345678*.

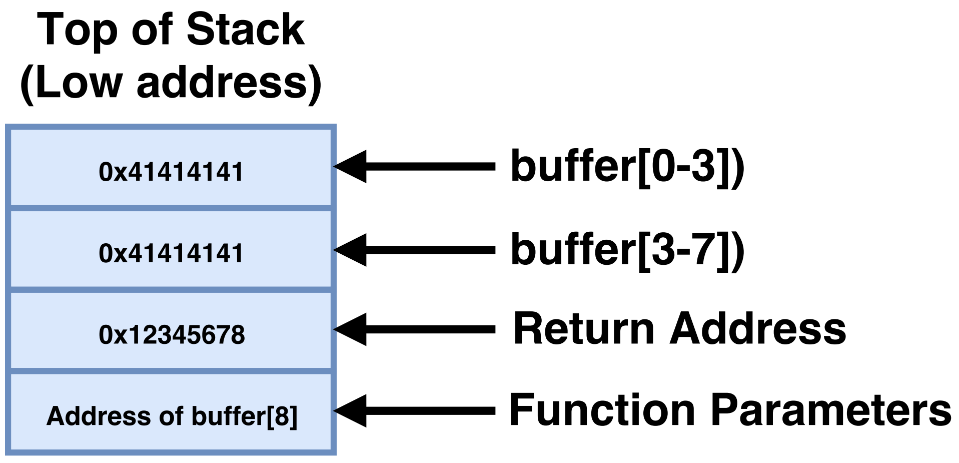


Figure 4.3: Stack Overflow.

Oftentimes, attackers will craft an exploit string that contains executable code, and overwrite the return address to point back to the exploit string. The exploit code can contain any *payload*, but often the goal is to initiate a connection to an attacker controlled machine and open a shell, giving full system access to the attacker. Note that many systems disallow stack code execution, thus thwarting this type of exploit, but do not prevent buffer overflows in general.

4.4 Replay Attack

Replay attacks are a category of exploit where an adversary intercepts or sniffs network traffic with the purpose of re-transmitting or delaying information [42]. Figure 4.4 illustrates a basic form of a replay attack. Suppose that Bob wishes to login to a network server. To do so, his encrypted password P_e is transmitted to the server over a network connection, where the server can then decrypt and authenticate the password. Eve sniffs (eavesdrops) the network and saves a copy of the encrypted password. Now, at any later time, Eve can send this encrypted password to the server and successfully login. In this simplified scenario, the server has no way to know if the password came from Bob or Eve. Note that Eve never even has to learn Bob's password – simply owning a copy of the encrypted version is enough to impersonate Bob.

4.5 Return Oriented Programming

Return oriented programming (ROP) is a type of attack that utilizes buffer overflow vulnerabilities (or any other attack that lets an attacker overwrite the call stack) to execute sequences of code already present on the system [43]. Through careful manipulation of the stack, an attacker can chain together small sequences of existing instructions (called "gadgets") to form arbitrary code sequences. A gadget generally consists of a few instructions, followed by a return statement.

For example, imagine a smart-lock system where a door lock is controlled by an Internet connected embedded device. We define the system with the following characteristics:

1. Function parameters are passed through registers R0, R1, R2, and R3.
2. The system stack follows the format of Figure 4.2, i.e. a function stack frame contains (from bottom to top) function parameters, return address, then local variables. The system word size is 32 bits.

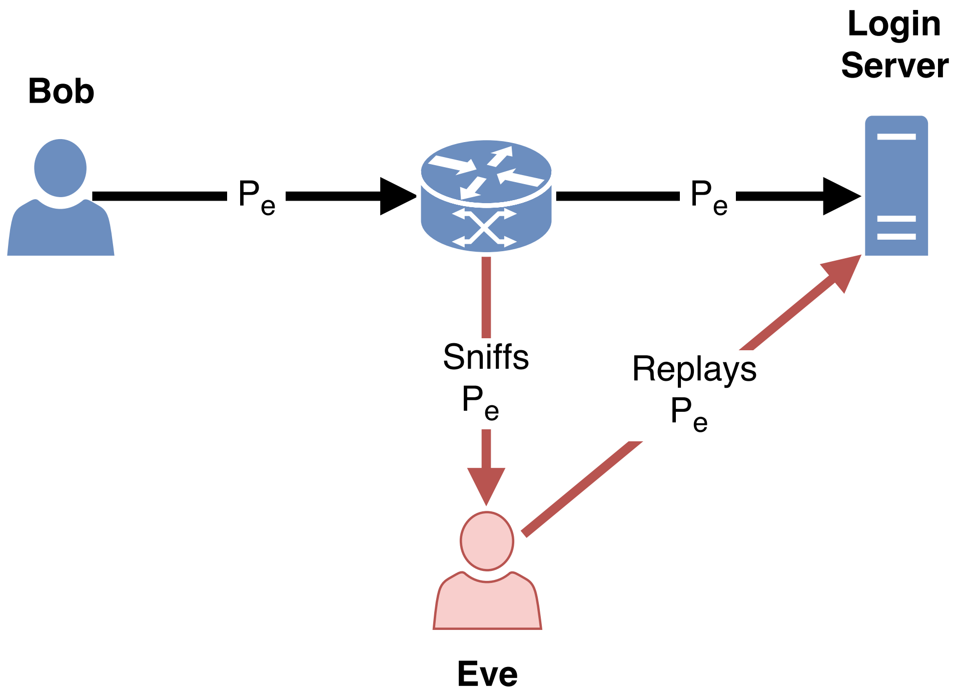


Figure 4.4: Replay attack. Eve captures Bob's encrypted password P_e and replays it at a later time to impersonate Bob.

3. The stack is non-executable.

4. The device code contains a function, *control_lock*, that is used to unlock or lock the door. Its location in memory is 0x10017C50.

5. The device code contains the buffer overflow vulnerability from listing 4.1.

6. Input is passed to the vulnerable function through external user input.

7. RET instructions pop the top value off the stack and program control continues at the address that has been popped.

Listing 4.2 shows the code for the *control_lock* function. Notice that if the integer value 1 is passed to the *control_lock* function, the door will be unlocked, whereas an integer value 0 will lock the door.

```c
void control_lock(int unlock)
{
  if(unlock == 1)
  {
    //Code to unlock door
  }
  else
  {
    //Code to lock door
  }
}
```

Listing 4.2: Smart Lock Control Function

Before diverting control to the *control_lock* function, an attacker must first place the value 1 in R0. Because of the non-executable stack, this must be done with ROP techniques. Using a gadget finder tool, we find two “chainable” gadgets. Listing 4.3 shows the example output of the gadget finder tool.

```
0x1001050C: MOV R7,#1; RET
0x10015144: MOV R0,R7; RET
```

Listing 4.3: Gadget Finder Tool Sample Output

To place a 1 in R0, we must chain the first gadget with the second. To do this, we need to first exploit the buffer overflow function to redirect program execution to the first gadget at address *0x1001050c*. In addition, the exploit string must contain addresses for both the second gadget and the *control_lock*. Therefore, our exploit string will be a sequence of byte values representing AAAAAAAA1001050C1001514410017C50. See Figure 4.5 for a depiction of the stack after the buffer overflow has occurred. Recall from Section 4.3 that the first eight ’A’ characters are simply to fill the local variable buffer *buffer[8]*.

Figure 4.6 shows a detailed view of the system state for each instruction of the exploit. Note that the instruction pointed to by an arrow during each timestep is the *next* instruction to be executed. With our exploit string, the value of 1 is stored in R0 right before program control is

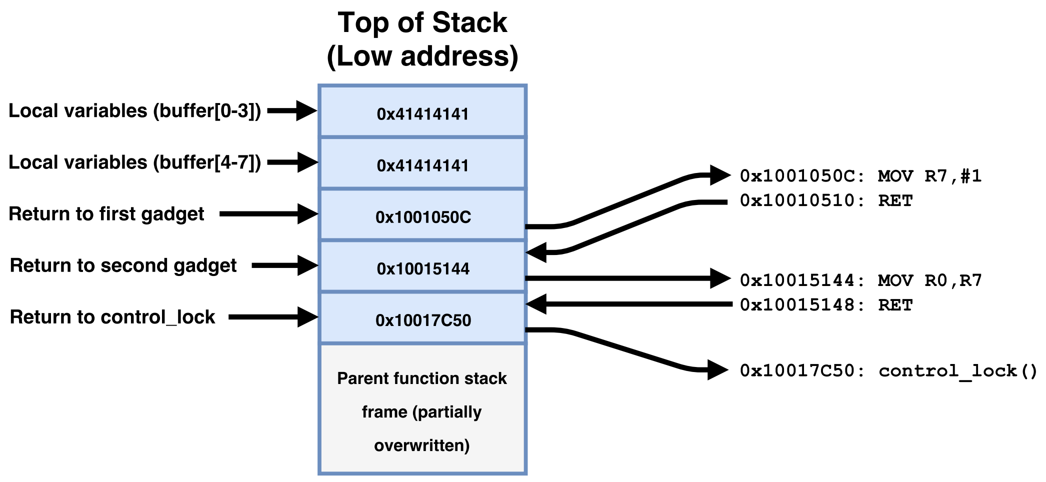


Figure 4.5: Return oriented programming gadget chain.

redirected to address *0x10017C50* (the address of *control_lock*). Thus the attack is successful in unlocking the smart lock.

From the example, one can see how an attacker could potentially chain together a much longer sequence of instructions. It has been demonstrated that a sufficiently large program contains enough gadgets to construct a Turing Complete library of gadgets, meaning that any algorithm can be constructed [44], [45]. Several tools, such as ROPgadget [46], exist to help locate and construct a library of gadgets. However, tools like ROPgadget require that an attacker have a copy of the executable code.

Return oriented programming attacks have become popular because it has become increasingly common for systems to employ non-executable stacks. Several techniques to mitigate ROP attacks have been suggested. Some use compiler techniques to create gadget-less binaries by instrumenting the code with special routines that only allow a RET instruction to execute if the function in which it resides was run from its starting point [47]. Another defense mechanism, ROPdefender, works by storing a copy of every function call return address on a shadow stack. When a return instruction is executed, the top value on the shadow stack is checked to ensure control is resuming at correct location in memory [48].

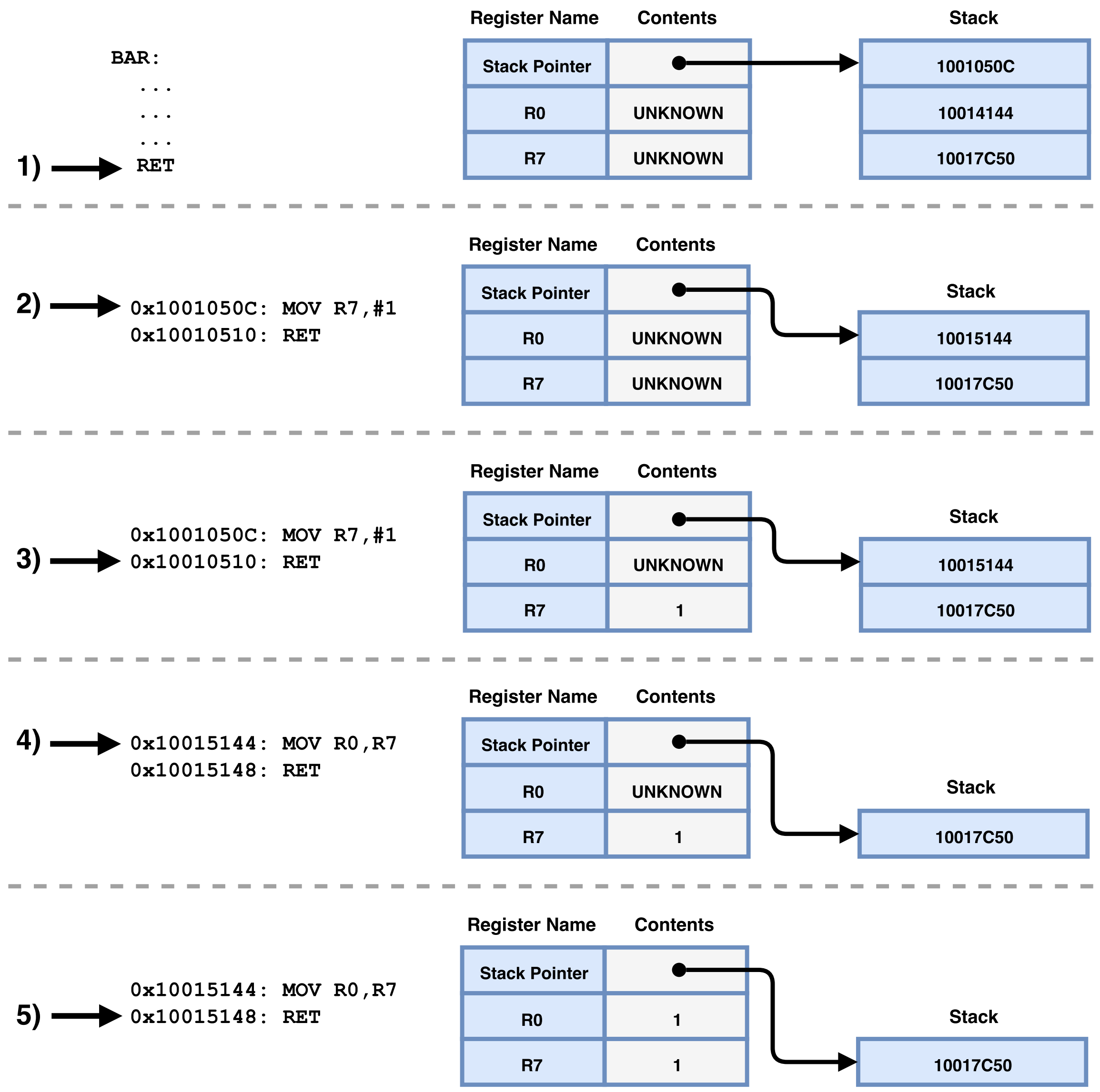


Figure 4.6: Return oriented programming sequence showing stack and registers at each step.

CHAPTER 5. REMOTE ATTESTATION FRAMEWORK

In this chapter, we present a remote attestation framework that builds upon several of the techniques discussed in Chapter 3. We show how the framework improves upon existing solutions by meeting the following goals:

- Improved capability for detection of malicious tampering.
- Increased flexibility – suitable for various embedded systems with different hardware components.
- Reduced system overhead.

We first present a discussion of several weaknesses in existing attestation methods that motivated the work of this thesis, followed by the description of the framework.

5.1 Shortcomings and Strengths of Current Methods

5.1.1 Memory Read Latency

A key component in many attestation schemes is some type of memory verification, where memory contents of a prover device are compared against a golden memory. A difference between the prover memory and golden memory indicate that the prover memory has been corrupted, perhaps by malicious tampering.

In most attestation protocols, a prover will condense its memory contents into a fixed-length representation, usually consisting of a hash or checksum. Checksum functions are similar to hash functions in that they take a variable length input and produce a fixed length output, but their purpose is to *check* data integrity. Ideally, a checksum function will exhibit the following properties:

1. It is easy to compute.
2. It takes input of arbitrary length, and produces a fixed-length output.
3. A small change in input will result in a significant change in output.

As opposed to transmitting an entire copy of device memory, the prover need only transmit the 128-bit checksum. It is possible that a system may have many legal memory configurations, each of which must be maintained by the verifier as a golden representation. In a system with n legal memory configurations, the verifier must either store n golden checksums/hashes or n copies of memory.

For attestation protocols utilizing software-only techniques, the time required to read memory can present significant overhead to the system, which can greatly degrade system performance. These effects are analyzed in Section 6.1.1. A more desirable technique is *direct memory access* (DMA) – special circuitry that allows read and write access to memory without the aid of the processor.

5.1.2 Nondeterministic Network Latency

Some attestation techniques such as SWATT rely on prover response time in the challenge-response protocol to detect tampering [13]. For example, imagine that we have an attestation protocol where a response is expected in 1 millisecond or less after the challenge is issued. If the prover takes longer than 1 millisecond, the verifier will assume that the prover has been tampered with and is expending extra time to forge a proof.

In theory, and under specific conditions, this can be a valid approach. However, remote attestation is most useful when it works over an Internet connection. Network latency is unpredictable because of the non-deterministic path that data may take to reach its destination. Each router in the path determines the next "hop" based on current traffic loads and various other factors.

To illustrate, imagine we have the prover-verifier attestation system depicted in Figure 5.1. The prover and verifier are physically separated, but connected over the Internet through a network of routers. Sometimes, communication between the verifier and prover occurs by data packet

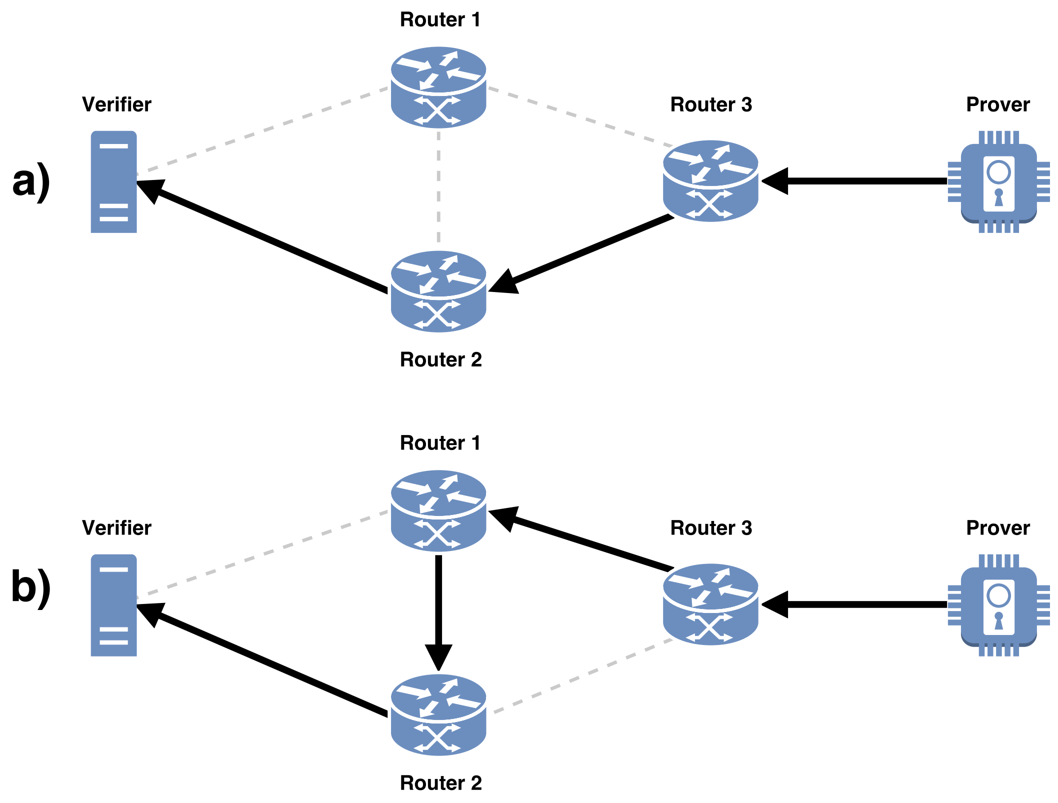


Figure 5.1: Internet packets can take different paths to reach the same destination. In a), the prover response only takes two hops to reach the verifier, whereas in b), three hops are taken.

transmission through routers two and three. At other times (perhaps due to congestion on the link between routers two and three), as depicted in b), data packets may travel through routers one, two, and three. We say that the data made two and three hops, respectively.

Even when the path is the same, the latency is non-deterministic. To physically demonstrate the nondeterministic nature of network latency, we used the Linux Traceroute utility to track the exact IP address of every router the data packets traverse to reach Facebook.com. To maintain privacy we do not list the IP addresses, instead we simply list the number of hops. Table 5.1 shows the results for five successive Traceroutes to Facebook.com from a single computer.

To properly interpret the results in Table 5.1, one must understand the basic functionality of the Traceroute utility. Every IPv4 and IPv6 packet header contains a *time to live (TTL)* field, sometimes referred to as the *hop limit*. The TTL field specifies how many hops a packet is allowed to take before it should be discarded. When data passes through a router, the TTL field is decre-

Table 5.1: The latency to reach Facebook.com for five successive Traceroutes. The latency varies by as much as 0.21 milliseconds to receive a response from the server.

	Hop number and time (ms) to reach										
	1	2	3	4	5	6	7	8	9	10	11
Request 1	0.29	0.59	0.27	0.56	0.74	3.57	19.91	18.76	18.57	17.65	17.63
Request 2	0.30	0.26	0.28	0.58	0.75	3.73	19.18	18.77	18.74	17.77	17.62
Request 3	0.31	0.32	0.31	0.56	0.74	3.72	18.73	18.65	18.58	17.61	17.52
Request 4	0.25	0.30	0.29	0.56	0.77	3.69	20.25	18.50	18.50	17.50	17.46
Request 5	0.28	0.31	0.26	0.55	0.77	3.73	21.61	18.91	18.73	17.73	17.67

mented by one. A router that encounters a packet with a TTL of zero will discard it and send an error message to the original sender.

Traceroute iteratively sends packets with incrementing TTL values, starting with a TTL of one. For each error message received, Traceroute can determine the round trip time (RTT) to and from that respective router (or hop). Packets continue to be sent until no more error messages are received, indicating that the packet has successfully reached its destination. Using this information, Traceroute constructs an estimate of the network path. Because information for a single hop is determined by a different packet, the results may not necessarily be indicative of the true path – and latency – of a single packet. Despite this, the results still provide us with an informative and useful estimate of the path.

The results shown in Table 5.1 indicate that the path taken by the five successive requests seem to take the same path each time. This is revealed by the consistency in relative time difference between hops for each of the Traceroutes. We see that the network latency to reach Facebook.com is nondeterministic, even on the same network path. It is possible that the path could change at a later time or date, thus inducing more variance. This is problematic for attestation protocols requiring strict timing of prover responses, because it requires the verifier to accept a wide range of response times, potentially allowing a compromised device to hide malicious activity in the extra slack.

Even if tight timing bounds could be guaranteed from one location, the response time will vary wildly between physical locations. Imagine a mobile phone application that takes the role of the verifier and the prover is a smart lock system. As part of the process to check the status of the smart lock, an attestation sequence is required. A user may wish to check the status of his/her lock from any physical location. Therefore an attestation protocol that relies on tight timing bounds over a network appears to be impractical due to network latency variance.

5.1.3 Executable Based Proofs

Many attestation protocols rely solely on the binary executable code in proof construction. Usually the proof is generated by taking a hash or checksum of the binary executable. As will be shown in Section 6.1.1, reading the entire memory may be costly, therefore time may be saved by reading only the executable code. This is especially true in systems that have large portions of unused memory; if the executable code fills up most of memory, not much computation time will be saved.

The issue here is that if an adversary is able to insert malicious code into some unused portion of memory, executable-based attestation will not detect the malicious code; these types of attestation can only detect tampering of the executable itself. Many types of malware (especially Botnet malware) are infact designed to leave the host system intact so as to go unnoticed. Malicious code can find its way onto a system through a variety of means such as buffer overflows, return oriented programming, exploitation of default credentials, software vulnerabilities, etc.

Based on this observation, we claim that executable-only attestation proofs are limited in their security guarantees. Our proposed solution overcomes this problem without requiring a prover to read out the entire memory layout.

5.2 Proposed Framework

As stated in the opening paragraph of this chapter, one of the design goals of our proposed attestation architecture is flexibility. In this section we present the framework for our attestation protocol, then show how the protocol can be adapted to suit different types of devices. More precisely, we show how it can be adapted for systems with built-in DMA capability, and also for devices without DMA capability.

5.2.1 Architecture

Our proposed attestation framework follows the challenge-response protocol presented in Figure 2.1. The prover device is divided into two domains: the secure and application domains (see Figure 5.2). The application domain consists of a processor, memory, peripherals, etc., and the secure domain is a hardware root of trust, implemented in an FPGA. The application domain is where all normal system functionality resides. In other words, it contains the application program and data. We assume that the application domain can be under full adversarial control.

The secure domain contains the functionality responsible for taking system measurements and constructing attestation proofs. This functionality is wrapped up in a logic block we call the *security monitor*. The security monitor contains a private encryption key (inaccessible to the application domain) that is used in the creation of its proofs. The cryptographic nature of the challenge/response protocol means that it is infeasible for a compromised application domain to forge an acceptable proof.

Because hardware is difficult (if not impossible) to modify, we can build assurance that the secure domain will boot into a trusted state. It is possible that an adversary may be able to modify the FPGA bitsteam or re-program the FPGA with a compromised bitstream, but this is beyond the scope of this paper. Methods to prevent bitstream reversal and tampering through encryption and various other methods have been proposed [17], [49], [50].

In order to initiate an attestation sequence, the verifier must communicate with the secure domain. To do so, we build upon XEBRA [15] and FPGA-Based Remote-Code Integrity Verification [17], where the verifier communicates with the secure domain indirectly as explained in Section 5.2.7.

5.2.2 Security Status

Most attestation protocols only perform a system measurement when initiated by a verifier. The protocol presented here differs in that regard; system measurements are taken at a periodic

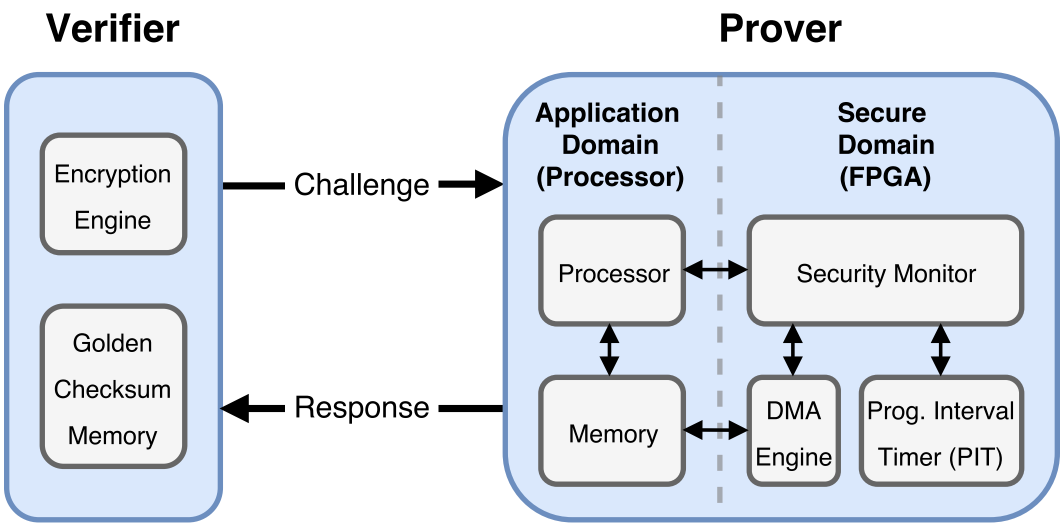


Figure 5.2: Top level remote attestation architecture.

interval as controlled by a programmable interval timer (PIT) (see Figure 5.2). The security monitor module is responsible for taking system measurements (described in Section 5.2.3) and uses them to determines the "security status". The security status can take on two values: "PASS" and "FAIL", which indicate the following:

- PASS - System is in a known good state.
- FAIL - System is in an unexpected state (corresponding to device compromise).

Measurements continue to be taken and security status updated without intervention of the verifier. Thus, when the verifier issues an attestation challenge, the prover simply reports the current security status to the verifier. As a result, the actual attestation process between a verifier and prover is very quick.

5.2.3 System Measurements

The security monitor takes two different measurements:

1. Checksum of executable in memory.
2. Current program counter (PC) value.

As is common in many attestation protocols, our protocol takes a checksum of the executable in memory. We show how this can be accomplished both with systems featuring DMA and those without in Sections 5.2.4 and 5.2.5 respectively. As discussed in Section 5.1.3, we claim that executable-only proofs are not sufficiently secure as malicious code in unused portions of memory will go undetected. To overcome this problem, our framework also probes the current PC value to verify that it is contained within a legal set of values, $\mathbf{PC_{legal}}$.

To understand how $\mathbf{PC_{legal}}$ is established, a basic understanding of linkers and linker scripts is required. To convert source code into the binary executable format that can be loaded and read by an embedded device, two major steps are required. First, the source files are compiled into object files. Second, all the object files are passed through a linker which combines (links) all of the object files and any library files into a single object file. Additionally, the linker must determine the addresses of the target device at which the code should reside. This is accomplished with a linker script, which allows the programmer to specify where the code should be placed in device memory.

To establish $\mathbf{PC_{legal}}$, we can simply examine the linker script to find the starting address of the executable (E_{start}). The size of the executable (E_{size}) can be determined by examining the executable headers. We can calculate the minimum PC value (PC_{min}) and maximum PC value (PC_{max}) as follows:

$$PC_{min} = E_{start} \tag{5.1}$$

$$PC_{max} = E_{start} + E_{size}. \tag{5.2}$$

Therefore the legal set of PC values ($\mathbf{PC_{legal}}$) is defined by:

$$\{x \in PC_{legal} \mid x >= PC_{min} \wedge x <= PC_{max}\}. \tag{5.3}$$

Note that equation 5.3 only holds if system code and application code are all placed in a single contiguous memory region. The framework can easily be adapted to accommodate devices where system code and application code are non-contiguous by adapting equation 5.3. It is also important to note that this method for determining $\mathbf{PC_{legal}}$ only works if the location of a program in memory is static. A system with an operating system will make it difficult to determine the exact location of the executable, because it could get loaded into differing parts of memory across different runs.

With these measurements (a checksum and PC value), the security monitor can then determine the current security status. The security monitor does this by comparing the memory checksum against the golden checksum, then verifies that the current PC value is a member of $\mathbf{PC_{legal}}$. If either of these two comparisons return false, the security status is set to "FAIL".

To summarize, the security monitor does the following when initiated by the PIT. The security status starts out in the "PASS" state.

1. Compute a memory checksum.
2. Probe PC.
3. Compare memory checksum against golden checksum.
4. Verify that current PC value is a member of $\mathbf{PC_{legal}}$.
5. If either step 3 or 4 fail, the security status is set to "FAIL". If failure occurred because of an invalid PC value, record the failing address. Otherwise, if failure occurred because of an invalid checksum, record the failing checksum.

Note that our protocol differs from most others in that the checksum comparison is done by the prover (in the secure domain as part of the measurement process), rather than the verifier. Thus, system characterization need only happen once (described in Section 5.2.6) and the verifier does not need to maintain a golden checksum. In this way, the secure domain acts as a sub-verifier, which is responsible for verifying internal state, then relaying this information to the true verifier.

The ideal period for the PIT, and therefore how often system measurements are taken, is explored and analyzed in Chapter 6.

5.2.4 DMA-Checksum

In this section we explain DMA-Checksum, a method for computing a memory checksum in systems where the security monitor has direct access to memory. As shown in Figure 5.2, the

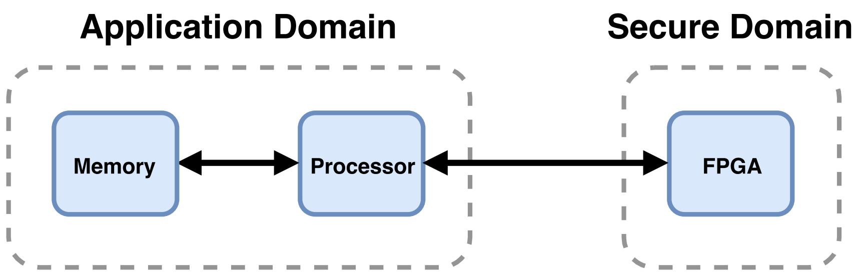


Figure 5.3: An example system where the secure domain does not have direct access to the application memory.

security monitor uses an FPGA DMA module to read memory. An XOR checksum is iteratively generated by reading the executable one word at a time and performing an XOR operation with the checksum for each read.

This is the ideal method for checksum creation, because it will produce little or no overhead to the application domain processor. However, DMA-Checksum only works if the security monitor is able to *directly* read memory. In other words, if DMA must be initiated by the processor, we cannot make accurate security claims. This is because if a compromised application must first be notified that we wish to initiate a DMA transfer, then the application can simply redirect the DMA transfer to a stored good copy of the code.

Therefore, this method of generating the checksum is limited to systems where the secure domain has direct access to the memory where the executable is stored. Many system-on-chip (SoC) devices contain this feature, so it is reasonable to assume that this approach is viable for modern devices.

In the next section, we show how a memory checksum can still be generated by systems where the secure domain does not have direct access to the device memory.

5.2.5 PRT-Checksum

In some systems, the secure domain may not have direct access to the device memory. For example, 5.3 illustrates a system where processor and memory are in one physical package, and the FPGA in another. In this specific example, the secure domain does not have direct access to memory.

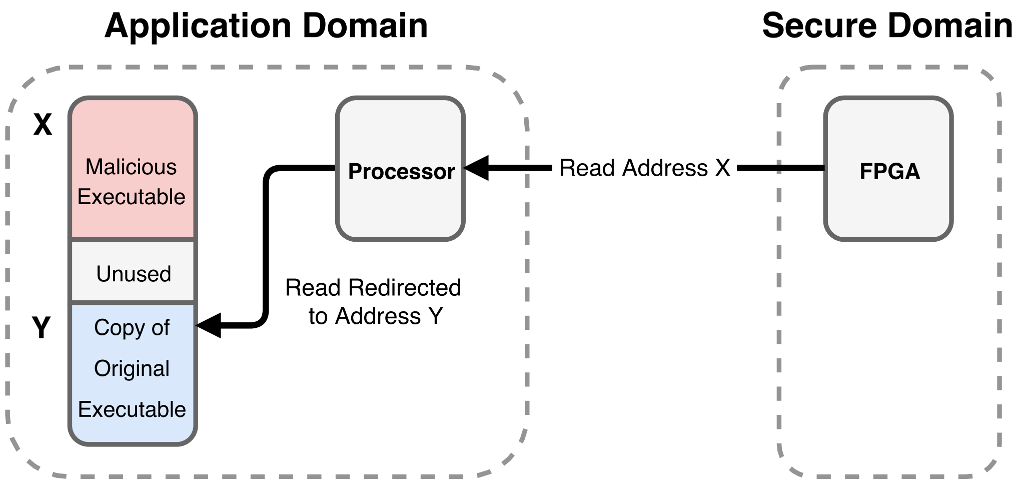


Figure 5.4: Example load redirect. A malicious application can redirect memory loads to a saved copy of the original executable.

To read memory in a secure manner, we build upon the techniques of SWATT (see Section 3.1.2) to provide a pseudo-random traversal checksum technique we call PRT-Checksum. PRT-Checksum differs from SWATT only in that timing is tightly controlled and measured between the security monitor and application domain, rather than measuring timing between the verifier and prover.

The scenario that PRT-Checksum is designed to prevent is illustrated in Figure 5.4. The secure domain cannot simply ask the application domain to provide it with the contents of a memory location, because a compromised application can easily redirect the load to a saved copy of the original executable. We refer to this as *load-redirection*.

The idea behind SWATT (and PRT-Checksum) is that in order to redirect a memory load, an extra if-statement (possibly involving multiple comparisons) must be added to the code to check if the memory location requested by the secure domain touches a compromised region, thus resulting in extra clock cycles. To prevent a compromised attacker from pre-loading a memory address in an attempt to thwart the strict timing requirements, a pseudo-random traversal of memory is performed such that a compromised application cannot predict which address will be loaded. The trade-offs for various traversal lengths are explored in the SWATT paper, but in our framework we choose the traversal length to be the size of the executable code (E_{size}).

When prompted by the PIT, the security monitor requests a checksum from the application domain as illustrated in Figure 5.5.

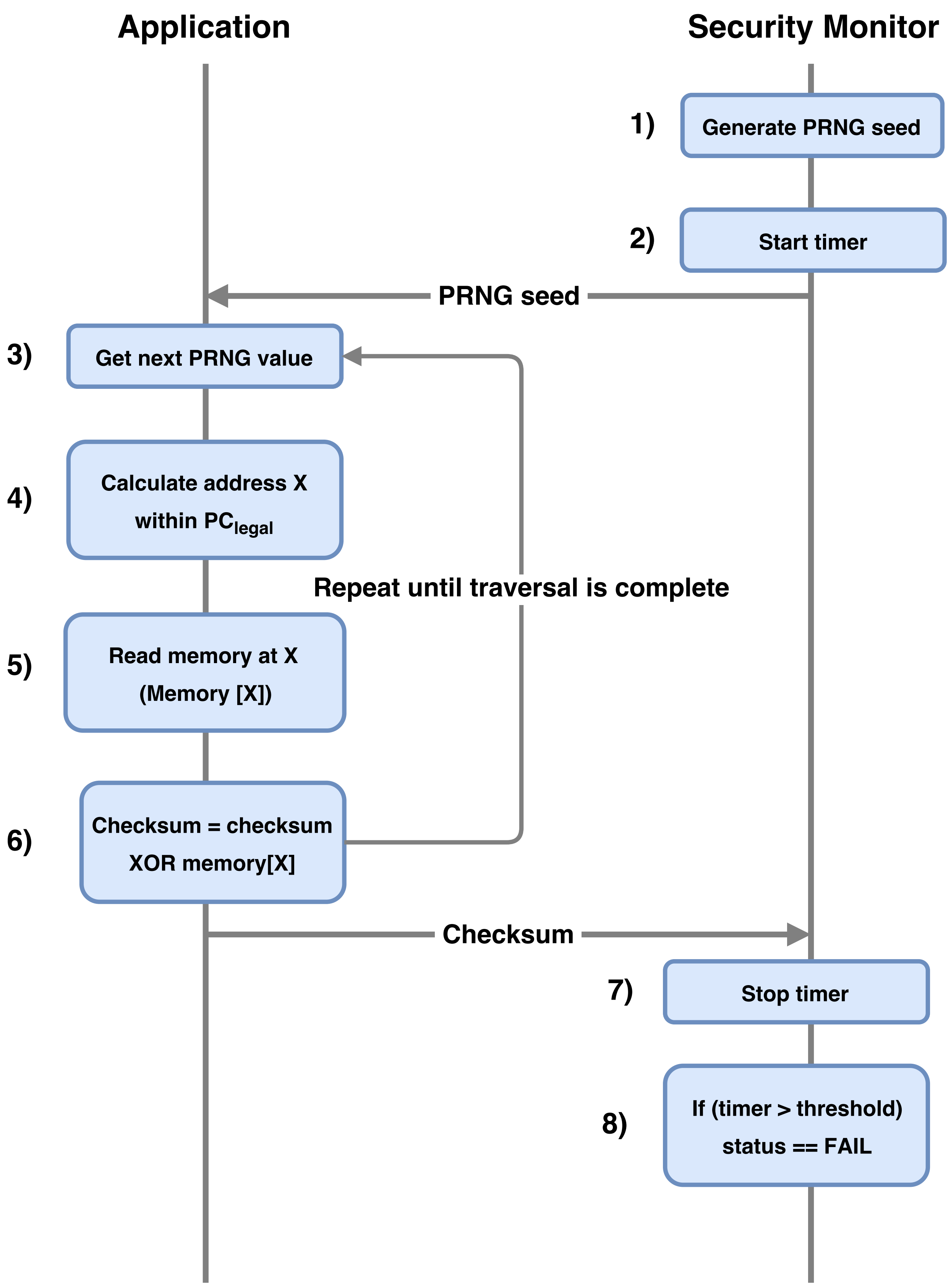

Application
Security Monitor
1)
Generate PRNG seed
2)
Start timer
PRNG seed
3)
Get next PRNG value
4)
Calculate address X within PC$_{legal}$
Repeat until traversal is complete
5)
Read memory at X (Memory [X])
6)
Checksum = checksum XOR memory[X]
Checksum
7)
Stop timer
8)
If (timer > threshold) status == FAIL

Figure 5.5: Checksum generation process for a system without DMA capability.

1. Security monitor generates a PRNG seed for the traversal.

2. Security monitor starts a hardware timer, then passes the PRNG seed to the application.

3. Application gets the next value from the PRNG.

4. PRNG value is used to calculate a random address X in the set $\mathbf{PC_{legal}}$.

5. Application reads memory at address X (Memory[X]).

6. Update checksum by performing $checksum = checksum \oplus memory[X]$. Repeat steps 3-6 until the specified number of iterations is complete (E_{size} in this case). When complete, send the checksum to the security monitor.

7. Security monitor stops hardware timer.

8. If the timer count value is too high, set security status to “FAIL”.

At step 5, a compromised application may attempt to redirect the memory load to address Y instead of address X. To do this, the application would have to perform an extra if-statement to see if address X is a modified memory location. Alternatively, the application could simply add an offset to address X, but in either case, additional instructions would be required. This would be detected as extra cycle time in the hardware timer kept by the security monitor. In other words, if the application takes too long to respond, the security monitor will assume that the application is redirecting loads and the security status is set to “FAIL”.

Also note that the PRNG algorithm contained in the application domain used for checksum generation must be the same algorithm used on the verifier side. Otherwise, the traversal used to compute a golden checksum will not be identical to the actual traversal.

Using the protocol described in Section 5.2.2 would require the security monitor to store a potentially infinite number of golden checksums. Therefore, in PRT-Checksum, the protocol is modified in the following manner. The security monitor does not store copies of golden checksums, rather, the verifier is responsible for computing golden checksums and comparing them with those it receives. The prover need only save the latest computed checksum and the seed that produced it. Both of these values are given to the verifier as a response to an attestation challenge, after which the verifier can compute the golden checksum using the given seed and a saved clean copy of the executable. The security monitor is still responsible for monitoring the PC and application domain response timing.

With regards to response timing, PRT-Checksum overcomes the shortcomings described in Section 5.1.2 because we are not dealing with non-deterministic network latency. The timing between the security monitor and application domain can be tightly bounded and controlled.

5.2.6 System Characterization

Before system deployment, the device must be characterized so as to generate the golden checksums and $\mathbf{PC_{legal}}$. This is as simple as taking a checksum of the memory and calculating $\mathbf{PC_{legal}}$ as described in Section 5.2.3 before deploying the system. These are then programmed into both the security monitor and verifier.

5.2.7 Attestation Protocol

As mentioned earlier, communication between the verifier and the secure domain is facilitated by the application domain. Figure 5.6 illustrates the flow of communication between the verifier and secure domain.

Because our framework focuses on Internet connected devices, we assume that the application domain will be running some type of network stack to enable remote access and control. A network stack is the software that implements a networking protocol such as TCP/IP.

The decision to route communication through the application domain is not an arbitrary one. While it would certainly be possible for the secure domain to run its own network stack, we argue that is unnecessary because the application domain already contains this functionality and therefore would require duplication of resources. The application domain also acts as a firewall to the secure domain, as certain traffic can be filtered out.

Attestation challenges and responses are encrypted to prevent a compromised application domain from tampering with the request or response. Without encryption, an application domain under adversarial control may be able to modify or forge the attestation response. In our framework, the attestation challenge consists of a password, known to both the verifier and security

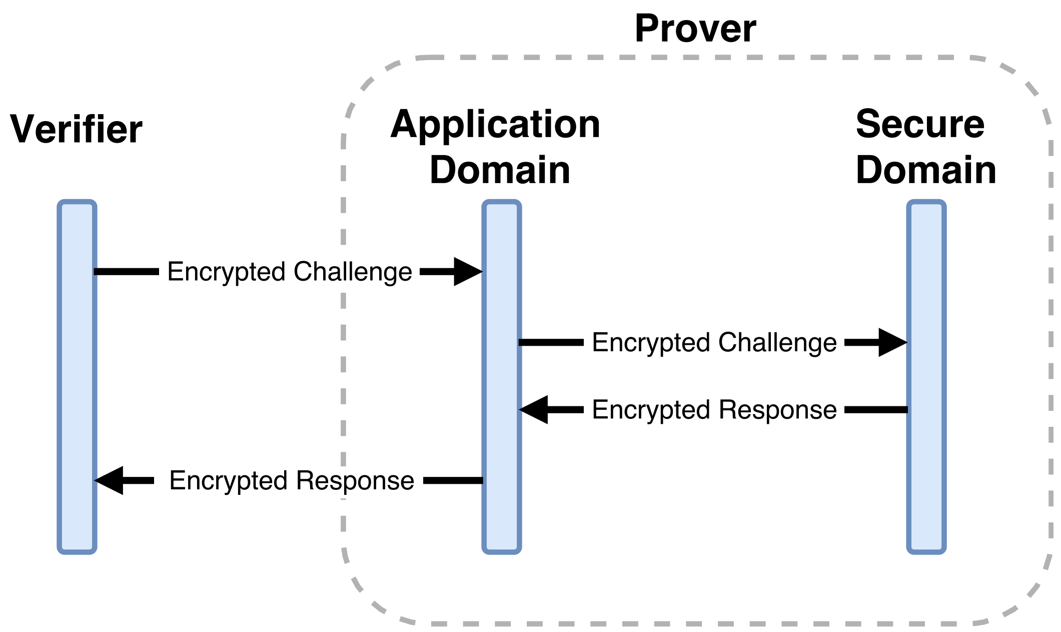


Figure 5.6: Attestation Protocol. Attestation challenges are passed to the secure domain through the application domain.

monitor, concatenated with a one-time session token (described in Section 5.2.8). The challenge is then encrypted with key K_s as follows: $K_s < Token, Password >$.

Similarly, the response contains a session token and password, but also contains the attestation status, checksum, and a PC address. If the security status is "FAIL", the PC value or checksum will correspond to either the PC value or checksum that caused the failure, respectively. Remember that when the security monitor detects an invalid PC value or checksum, it is recorded. Otherwise, if the status is "PASS", then these will correspond to the PC value and checksum from the last system measurement. For DMA-Checksum, the verifier does not specifically require the checksum, but in some circumstances it may be useful for a verifier to have the checksum. Additionally, it helps provide input randomness to the encryption process. The PC address is also not explicitly required for attestation, but is useful to a system administrator who may wish to know where the PC assertion failed. The response is encrypted with K_s as follows: $K_s < Token,$ $Password, Attestation\ Status, Checksum,\ PC\ address >$.

To perform encryption and decryption, both the verifier and the prover (specifically the security monitor) must possess the same symmetric encryption key K_s. A symmetric key encryption algorithm is one in which both encryption and decryption uses the same key; in other words both parties participating in the encryption/decryption process must have a copy of the same key. In our proposed framework, K_s must be pre-programmed into both the prover secure domain and the verifier device.

As shown in Figure 5.2, the verifier device must contain an encryption engine and the security monitor module must also contain an encryption engine. To initiate attestation, as shown in Figure 5.6, the verifier first encrypts the challenge, which is sent to the application domain, then forwarded to the secure domain. The secure domain decrypts the challenge, verifies that it comes from an authentic verifier (explained in Figure 5.2.8), then answers with an encrypted response by passing it to the application domain which forwards it to the verifier.

The verifier uses its copy of K_s to decrypt the response and then responds to the status accordingly. How the verifier responds to the security status is largely application specific. It might, for example, notify a user or administrator and halt future communication with the device.

It is possible that a compromised application may refuse to forward the attestation challenge or response and simply drop it. However, if this occurs, the verifier will never receive a valid response, and therefore assume device compromise. Using a sufficiently strong encryption protocol makes it computationally infeasible for a compromised application to forge or tamper with a response or challenge. Therefore, a compromised application has incentive to forward the challenges and responses. Failure to do so will be detected quickly by the verifier.

5.2.8 Preventing Replay Attacks

To prevent replay attacks, the challenge is tagged (before encryption) with a "nonce". A nonce is a one-time-use number that is used in cryptographic communication to ensure that the communication cannot be used in replay attacks. The nonce is concatenated with the data, then encrypted. Modern encryption protocols such as AES exhibit the property that a small change in input will result in an output that does not resemble the original output.

For example, suppose we wish to encrypt the word "Attestation" with AES-128 and send it across a network. We can prevent replay attacks by concatenating the output of a counter with the word "Attestation" and then encrypt. Table 5.2 shows the outputs of AES-128 using various nonce values. Notice that for each successive input, the message was modified only slightly, but

Table 5.2: AES-128 encryption of various inputs with a nonce using key 0x00001111222233334444555566667777.

Input Text (Plaintext)	**Encrypted Output Text (Hexadecimal)**
Attestation	`7a18f6c57a8420c6c875b4b5c5993343`
0_Attestation	`944f7d8a75e6c6766f1acbf3a408f2c2`
1_Attestation	`a4947e2fcc4b01d25231722035d0e9d2`
2_Attestation	`69ab00b57695a1e65c423b2f967b837d`
3_Attestation	`199fa4732f606b0e48feeff00139d05c`

the output changed greatly in each case. This property makes it computationally infeasible for an attacker to forge a message with the proper nonce.

The receiving party can then decrypt the message and verify that it has not been victim of a replay attack by checking the nonce. If the receiving party receives a message with the same nonce twice, it can be sure that the second message was a replay attack and the message will be ignored. In real-word applications, the nonce usually consists of a pseudo-random number.

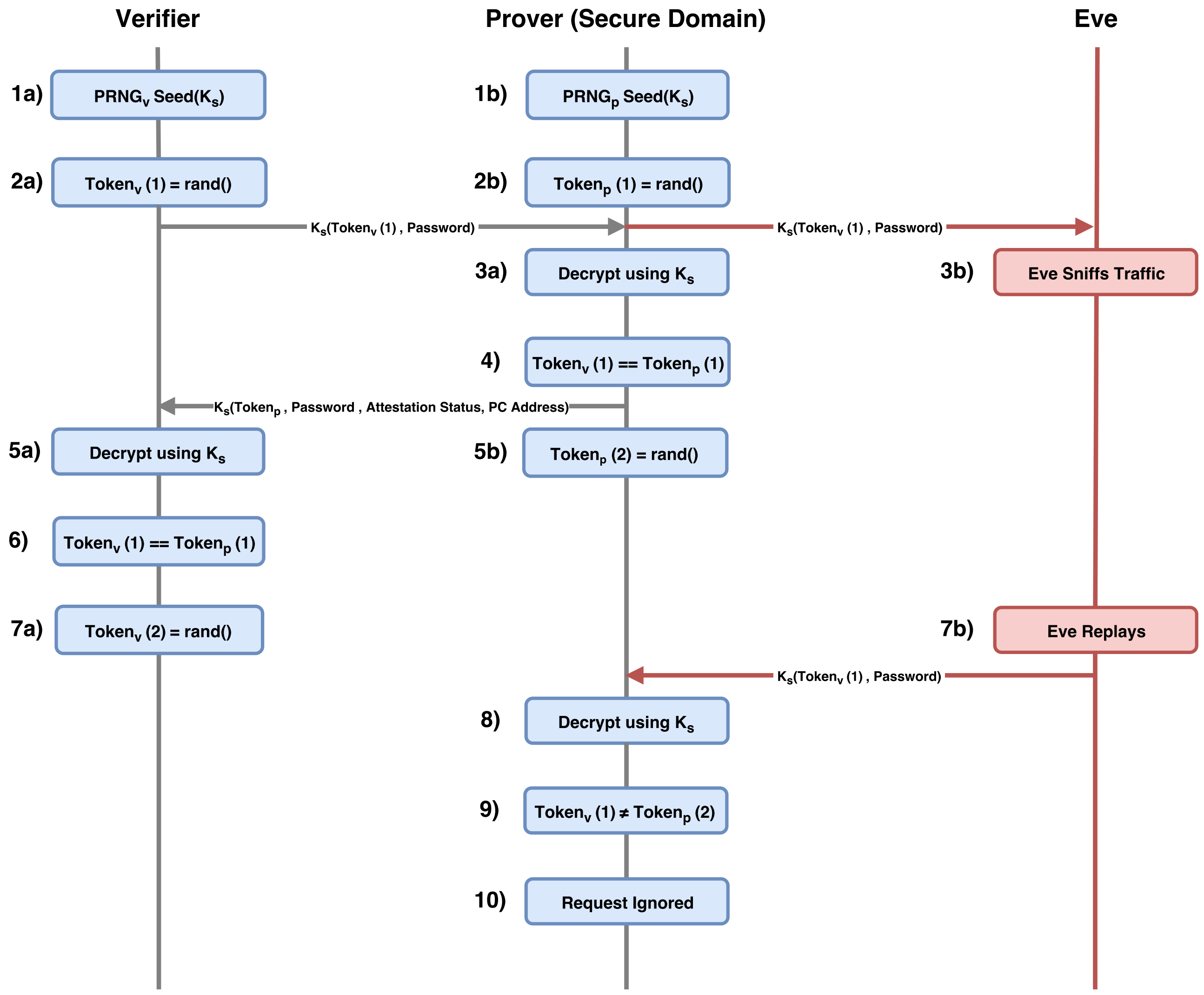
In our framework, we use a session token as a nonce to prevent replay attacks. The session token consists of a pseudo-random number, and is generated by both the verifier and the prover security monitor. Both the prover and security monitor must contain an instance of a pseudo-random number generator (PRNG) and seed it with K_s. It is imperative that both PRNG instances use the same pseudo-random algorithm and seed value. The verifier PRNG can be implemented in software, and the security monitor in hardware, so long as they both produce an identical stream of random numbers given the same seed.

Figure 5.7 illustrates the attestation protocol. An explanation of each step is given as follows:

1. PRNG initialization. This step need only occur once at system initialization.

   a) Verifier PRNG ($PRNG_v$) seeded with K_s.

   b) Prover PRNG ($PRNG_p$) seeded with K_s.



Verifier
Prover (Secure Domain)
Eve
1a) PRNG$_v$ Seed(K$_s$)
1b) PRNG$_p$ Seed(K$_s$)
2a) Token$_v$ (1) = rand()
2b) Token$_p$ (1) = rand()
K$_s$(Token$_v$ (1) , Password)
K$_s$(Token$_v$ (1) , Password)
3a) Decrypt using K$_s$
3b) Eve Sniffs Traffic
4) Token$_v$ (1) == Token$_p$ (1)
K$_s$(Token$_p$, Password , Attestation Status, PC Address)
5a) Decrypt using K$_s$
5b) Token$_p$ (2) = rand()
6) Token$_v$ (1) == Token$_p$ (1)
7a) Token$_v$ (2) = rand()
7b) Eve Replays
K$_s$(Token$_v$ (1) , Password)
8) Decrypt using K$_s$
9) Token$_v$ (1) ≠ Token$_p$ (2)
10) Request Ignored

Figure 5.7: Replay attacks are prevented by tagging each attestation request with a session token (nonce).

2. Generate session token. The number in parenthesis indicates the token number in sequential order for ease of reading, not necessarily the actual token value. Note that $Token_p$ and $Token_v$ are assumed to be equivalent. $Token_v$ and the password are encrypted with K_s, and then transmitted to the verifier.

   a) Verifier token ($Token_v$) is generated by getting the next random value in $PRNG_v$ stream.

   b) Prover token ($Token_p$) is generated by getting the next random value in $PRNG_p$ stream.

3. Challenge is received.

a) Prover receives the challenge and decrypts it using K_s.

b) Eve is sniffing traffic and saves a copy of the encrypted challenge.

4. Prover checks that $Token_p$ equals $Token_v$. The password is also checked at this step. If the tokens match and the password is correct, an attestation response is sent to the the verifier. The response consists of $Token_p$ concatenated with the attestation status (pass/fail), password, and PC address. The response is then encrypted with K_s.

5. The verifier receives the attestation response.

   a) Verifier decrypts the response using K_s.

   b) Prover generates a new token $Token_v$ by getting the next random value in $PRNG_v$ stream.

6. Verifier checks that $Token_p$ equals $Token_v$ and that the correct password is supplied. At this point, the verifier responds according to the attestation status. If attestation failed, the user might be notified or communication with the prover halted. The manner in which a verifier responds to an attestation failure is application specific.

7. a) A new verifier session token ($Token_v$) is generated by getting the next random value in the $PRNG_v$ stream.

   b) Eve replays her saved copy of $K_s < Token_v, Password >$ by transmitting it to the prover in the attempt to initiate an attestation sequence.

8. Prover decrypts the challenge K_s.

9. Prover compares $Token_v$ with $Token_p$ and finds that $Token_v$ is invalid, indicating a replay attack attempt.

10. Because of an invalid session token, the prover ignores the request and does not provide a response to Eve.

To preserve the integrity of the above protocol, it is essential that the PRNG algorithm and initial seed value remain secret, hence our choice for using K_s as the seed; K_s will already be kept secret.

5.2.9 Implementation

We implemented our attestation framework on a Digilent Zybo board. The Zybo board features a Xilinx Zynq Z-7010 SoC, which contains an ARM Cortex-A9 processor, Xilinx 7-Series FPGA, and onboard memory controller. The board also contains 512 MB of DDR3 memory, which interfaces with the Zynq chip via an Advanced Microcontroller Bus Architecture (AMBA) interconnect [51], [52]. Figure 5.8 shows a high level architecture of our implementation. We define the entire programmable logic (FPGA) portion of the chip as our secure domain, and the processing system as the application domain. Because the secure domain is implemented in an FPGA, the application domain cannot modify it, giving us a hardware root of trust.

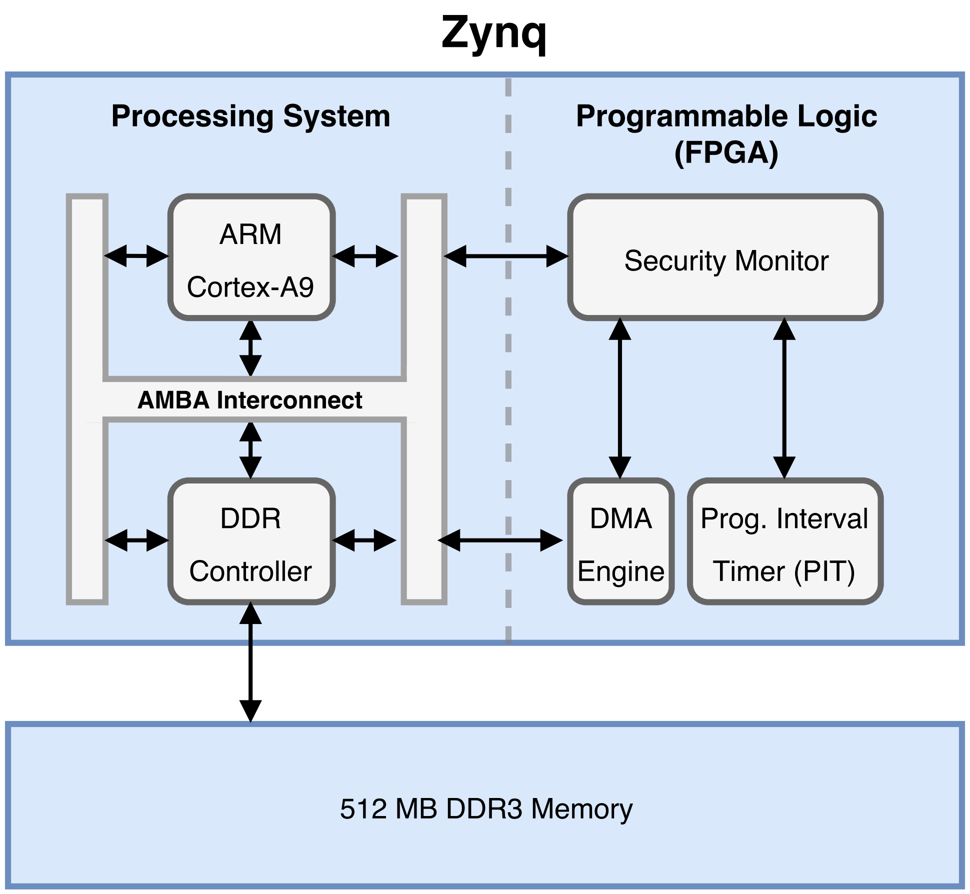


Figure 5.8: Remote attestation implementation on Zybo board.

The secure domain components, including the PIT, security monitor, and DMA engine were all written in Verilog. For encryption/decryption we used the open source "Tiny AES" module from

OpenCores.org [53]. Application and verifier software components were written in C, using the Xilinx SDK development environment.

The security monitor was implemented as an AMBA Master, meaning that it is allowed to initiate AMBA bus transactions. This is necessary so that the security monitor can initiate memory reads to the DDR controller. All communication between the application and the security monitor happens through a register interface.

The ARM processor does not provide direct access to the PC register. Therefore, to determine the PC value, we make use of the ARM interrupt mechanism. When an interrupt occurs, the processor must save the current context by pushing all of the register values onto the stack, and placing the return address in the “link register”. This is done so that when the ISR finishes execution, the processor can return to the same state it was in before the interrupt occurred, including the PC value where execution will resume. We utilize this mechanism by raising a hardware interrupt and inserting a assembly routine that executes before the ISR. The assembly routine saves the value of the link register, subtracts 4 (since the return address in the link register is the address of the *next* instruction to be executed), and then stores it in a global variable. The ISR then reads the global PC variable and writes it to the security monitor.

We ensure that the assembly routine and ISR described above have not been modified by taking the memory checksum before performing the PC probe interrupt. Thus, if the ISR or assembly routine have been modified, it will be manifest in the memory checksum.

To generate golden checksums, we created a utility called Checksum-Gen that is capable of creating both PRT and DMA checksums given an input binary executable. While our tool is capable of creating both types of checksums, only one type will be needed for a given system. As a reminder, PRT checksums are only required for systems with no DMA capability. The tool also determines the length of the .text section by examining the .elf headers. The .text section contains the application code, therefore we use this number for E_{size} in determining $\mathbf{PC_{legal}}$.

To determine $\mathbf{PC_{legal}}$, we use E_{size} from Checksum-Gen, then find E_{start} by manually examining the linker script that is generated by Xilinx SDK (the development environment we use for creating application domain software). Using these values, we can compute PC_{min} and PC_{max} and therefore $\mathbf{PC_{legal}}$.

CHAPTER 6. RESULTS AND ANALYSIS

In Chapter 5 we presented the framework for a remote attestation protocol and showed how it can be implemented both on systems with DMA capability and those without. In this chapter, we present our test results and analysis.

The tests performed in this chapter were run using our hardware implementation described in Section 5.2.9. For the software, we use a test application, consisting of a Tic-Tac-Toe game. This program was chosen because it utilizes many of the Zybo board system components. The program computes Tic-Tac-Toe moves using the minimax algorithm, which works by recursively computing a score for each potential next move, based on all possible game outcomes of that move. The move with the highest score is chosen as the next move. The algorithm is processor intensive, especially at the beginning of the game where the number of possible outcomes is large.

Not only is the test application processor-intensive, but it makes extensive use of an LCD display. Through DMA writes to a video buffer, the processor is able to display images to the LCD display. The moves for player 1 are randomly calculated, then player 2 responds by computing the next best move via the minimax algorithm. The program is roughly 164 KB in size.

6.1 Functional Results

In this section we present experimental results that further motivate several of the design choices made for our attestation framework. We then present the functional results of our attestation framework – using both the DMA-Checksum method and the PRT-Checksum method. For the functional tests presented here, the PIT interval is set to 50ms. Our desktop computer acts as the verifier device, and is connected to the prover via a serial connection. We use a serial connection for ease of test and proof-of-concept, but communication could just as easily be performed over an Internet connection. Also, for ease of test and demonstration of proof-of-concept, we are not performing encryption or decryption. TinyAES only requires 29 clock cycles to encrypt or decrypt, and since this only occurs when a challenge is issued by the verifier, the overhead induced by encryption/decryption is negligible.

6.1.1 Memory Read Time

To further motivate the design choice to take a checksum of just the executable and not the entire memory layout, we measure the time it takes our implementation to generate a checksum of various executable sizes. Figure 6.1 shows the results. For comparison purposes, we also measured the time required to generate a simple software checksum by sequentially reading memory and performing the XOR operation. These measurements are recorded in the figure and labeled "software".

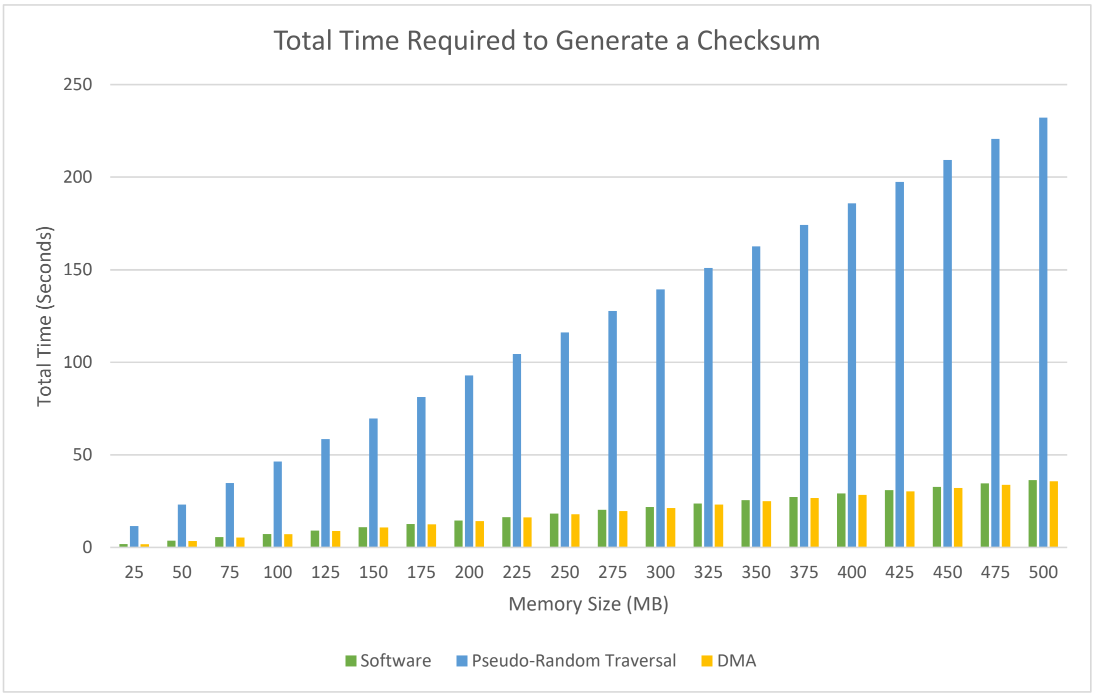


Figure 6.1: Memory (512 MB DDR3) read time for various read lengths.

As expected, PRT-Checksum adds significant overhead to the checksum process. The added overhead comes from the process of generating random numbers and performing checksum calculations. Our implementation used a linear-feedback shift register for generating random numbers, followed by a modulus operation to map the pseudo-random value to an address in the proper memory range. Perhaps an optimized PRNG algorithm and a computation without modulus or division would result in less overhead. Regardless, PRT-Checksum will always result in more overhead than DMA-Checksum.

As previously discussed, without probing the PC value, a checksum of the entire memory layout would be required to ensure that extraneous code isn't being stored in an unused portion of memory. Our results show that to generate a checksum of the entire 512MB memory for our implementation takes approximately 35 and 232 seconds for the DMA-Checksum and PRT-Checksum respectively. Thus, if there are large portions of unused memory, an executable-only checksum will result in much less overhead and latency.

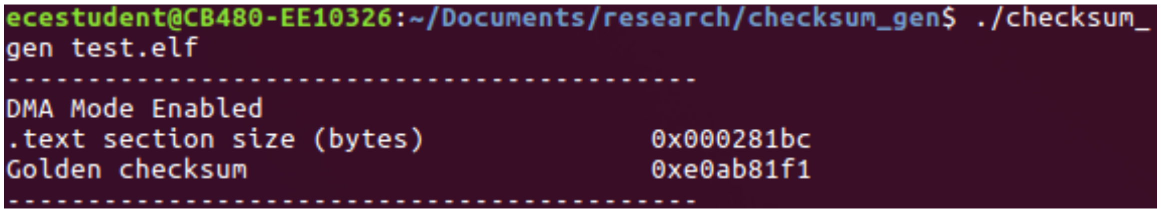
6.1.2 DMA-Checksum

We first test the functionality of attestation using DMA-Checksum. To start, we characterize the system in the following two steps:

1. Generate the golden checksum of the test program executable.
2. Calculate $\mathbf{PC_{legal}}$.

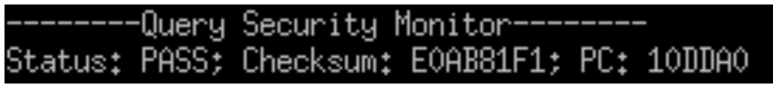
To generate the golden checksum on the verifier side, we use our Checksum-Gen tool with the test executable as input. Figure 6.2 shows the two outputs of the tool: the size of the .text section (the executable code), and the resulting golden checksum of the executable. As a reminder, the size of the executable (E_{size}) is used in calculating $\mathbf{PC_{legal}}$.

To calculate $\mathbf{PC_{legal}}$ we first examine the linker script to find E_{start}, which we find to be 0x00100000. Using E_{start} and E_{size} we compute PC_{min} and PC_{max} to be 0x00100000 and 0x001281BC respectively. Finally, we compute $\mathbf{PC_{legal}}$ with $\{x \in PC_{legal} \mid x >= PC_{min} \land x <= PC_{max}\}$.



```
ecestudent@CB480-EE10326:~/Documents/research/checksum_gen$ ./checksum_
gen test.elf
-------------------------------------------
DMA Mode Enabled
.text section size (bytes)              0x000281bc
Golden checksum                         0xe0ab81f1
-------------------------------------------
```

Figure 6.2: Output of the Checksum-Gen tool for a DMA-Checksum. The output is the golden checksum, as computed by providing the executable file as input.



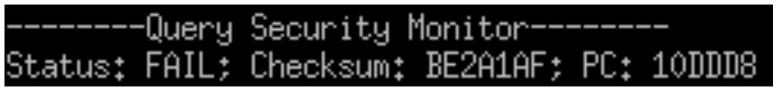
```
--------Query Security Monitor--------
Status: PASS; Checksum: E0AB81F1; PC: 10DDA0
```

Figure 6.3: DMA-Attestation results as shown by the verifier. The output consists of the prover response after initiation of an attestation challenge. This response reflects the system measurements of an unmodified executable.

The golden checksum and $\mathbf{PC_{legal}}$ are both programmed into the FPGA bitstream as part of the security monitor and loaded onto the Zybo board. We then initiate an attestation challenge via the serial console, and receive the output shown in Figure 6.3.

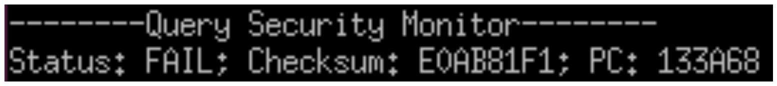
Remember from Section 5.2.7 that the attestation response contains a token, password, attestation status, checksum, and PC address. The output of our verifier program (shown in Figure 6.3) only shows the security monitor attestation status, computed checksum and PC value. The password and token are only used internally; if either are incorrect the verifier will display an error message. The "PASS" status indicates that the application memory is intact and the security monitor has not detected an invalid PC value. As a sanity check, we can manually verify that the most recently computed security monitor checksum and PC value (as given in Figure 6.3) are valid. Indeed, the security monitor checksum matches the golden checksum (as computed by Checksum-Gen in Figure 6.2) and the PC value is less than 0x001281BC and greater than 0x00100000. Remember that the attestation status is all the verifier requires to make security claims (in DMA-Checksum), but the checksum and PC value are included in the response because they may be of value to the verifier, as discussed in Section 5.2.7.

Next we load an executable with a slight modification – one line of altered C code. We load the modified code onto the Zybo board and initiate an attestation challenge via the serial console. The results of the attestation are shown in Figure 6.4. As expected, we see a "FAIL" attestation



```
--------Query Security Monitor--------
Status: FAIL; Checksum: BE2A1AF; PC: 10DDD8
```

Figure 6.4: DMA-Attestation results as shown by the verifier. The output consists of the prover response after initiation of an attestation challenge. This response reflects the system measurements of a modified executable.



```
--------Query Security Monitor--------
Status: FAIL; Checksum: E0AB81F1; PC: 133A68
```

Figure 6.5: DMA-Attestation results as shown by the verifier. The output consists of the prover response after initiation of an attestation challenge. This response reflects the system measurements of a system with extraneous code placed in unused memory.

status. The checksum and PC values help provide some insight as to why attestation fails. The PC value is greater than 0x00100000 and less than 0x001281BC, indicating that the processor has only been executing instructions that belong to $\mathbf{PC_{legal}}$. However, the checksum we received from the security monitor – 0x0BE2A1AF – does not match our golden checksum. Because the executable code was modified, the security monitor computed a different checksum, and we were successful in detecting altered code.

Finally, we ensure that the security monitor can properly detect malicious code running in an unused portion of memory. We load and run the original executable that has been augmented with additional code outside $\mathbf{PC_{legal}}$. Again, we initiate the attestation via the serial console and the resulting output is shown in Figure 6.5. Notice that the checksum in Figure 6.5 is correct, which is expected because memory contents did not change in the range of the original executable. However, the results shows that an invalid PC value (0x00133A68) was detected, therefore the security monitor shows “FAIL” status.

6.1.3 PRT-Attestation

To prove the validity of PRT-Checksum, we must ensure that we can detect load-redirects. To do this, we measure how long it takes the application to perform a memory read, then perform the same measurement with an added if-statement in the code. We perform this measurement with the hardware timer built into the security monitor to achieve cycle-accurate timing. The code for these two measurements is shown in listings 6.1 and 6.2. We performed 10 consecutive reads, each with a random address (to better simulate a pseudo-random traversal) and show the results in Table 6.1.

```
timer_clear();
timer_start();

read_data = Xil_In32(read_address);

timer_stop();
```

Listing 6.1: Memory Read Code

```
timer_clear();
timer_start();

if(read_address >= min && read_address <= max) {
  read_data = Xil_In32(read_address + malicious_offset);
}
else {
  read_data = Xil_In32(read_address);
}

timer_stop();
```

Listing 6.2: Memory Read Code With Added If-Statement

The average read time (in cycles) for the 10 memory reads was 142.4 using the original code (listing 6.1), and 159.4 with the added if-statement (listing 6.2). We observe that, on average, the addition of an if-statement incurs additional delay; therefore PRT-Checksum will be feasible on our system.

Table 6.1: The time required (in cycles) for 10 pseudo-random memory reads with and without an added if-statement.

	Read number and time (in cycles) taken									
	1	2	3	4	5	6	7	8	9	10
Original (no if-statement)	304	130	130	96	132	118	126	126	130	132
Added if-statement	290	146	146	112	156	134	160	154	154	142

To perform PRT-Checksum, we must first characterize the system in the following ways:

1. Determine $\mathbf{PC_{legal}}$
2. Determine acceptable application domain response time, R_{max}.

The method used to calculate $\mathbf{PC_{legal}}$ for PRT-Checksum is identical to the process used in DMA-Checksum. Because we are using the same test program, $\mathbf{PC_{legal}}$ will be identical to that of Section 6.1.2.

Next, we calculate the acceptable application domain response time, R_{max} – i.e. the maximum allowable time for an application to generate a checksum (steps 3 - 6 in Figure 5.5). We do this by characterizing the response time over 100 attestation challenges. Table 6.2 shows the minimum, maximum, and average response times of these 100 challenges. Additionally, we perform the same measurements on an application containing an added if-statement (similar to the one shown in listing 6.2) in the attestation code. We then define R_{max} as follows:

$$R_{max} = O_{max} + \frac{A_{min} - O_{max}}{2}, \tag{6.1}$$

where:

1. O_{max} represents the maximum recorded response time for the original code, and
2. A_{min} represents the minimum recorded response time for the code with the added if-statement.

The intuition behind equation 6.1 is that it is entirely possible that we did not measure the true A_{min} or O_{max}, and to reduce the probability of false positives or negatives, we compute R_{max}

Table 6.2: Prover response time characterization. These values correspond to the minimum, maximum and average times (in cycles) required for the attestation code to compute a checksum. Obtained over 100 runs.

	Minimum (cycles)	Maximum (cycles)	Average (cycles)
Original (no if-statement)	7422726	7470896	7452021.6
Added if-statement	7632450	7731428	7694039.7

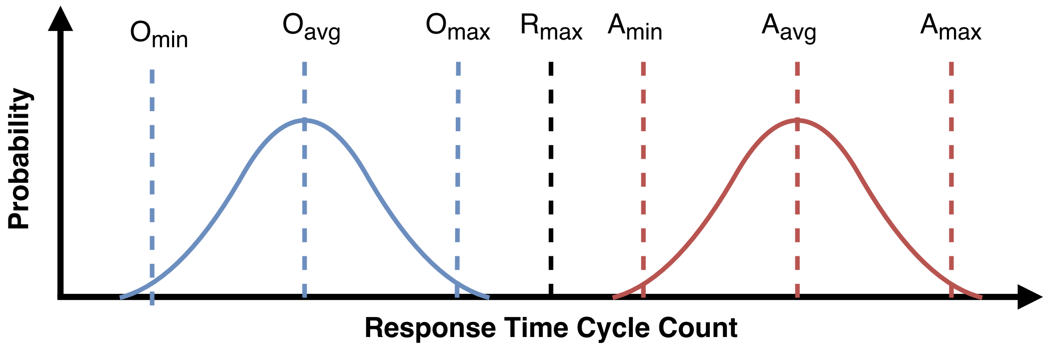
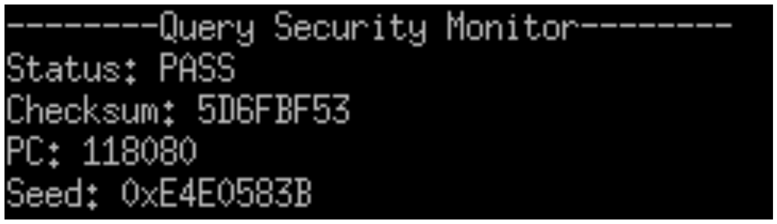


Figure 6.6: A conceptual depiction of the probability of various application domain response times for the original code (O) and the code with the added if-statement (A). We compute R_{max} such that it lies halfway between our measured O_{max} and A_{min}. This is done to minimize the likelihood of false positives and negatives from the security monitor.

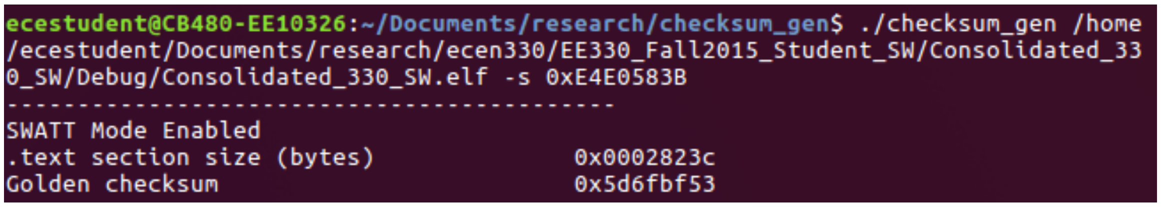
to be halfway between our measured A_{min} and O_{max}. See Figure 6.6 for a conceptual visualization of the response time probability and R_{max} selection. Note that Figure 6.6 does not represent the response time probability based on our measured response times; it is merely intended to aid in the understanding of the intuition behind equation 6.1.

To summarize, we set R_{max} to 7551673 as calculated by equation 6.1. Thus, if the application domain ever takes longer than 7551673 cycles to compute a checksum, then the security monitor will set the attestation status to "FAIL". We program R_{max} and $\mathbf{PC_{legal}}$ into the FPGA bitstream and load it onto the Zybo board. We load our unmodified executable and initiate an attestation request via the serial console. The response from the prover to the verifier is shown in Figure 6.7. The "PASS" status indicates that the secure domain has not detected load-redirection up to this point.



```
--------Query Security Monitor--------
Status: PASS
Checksum: 5D6FBF53
PC: 118080
Seed: 0xE4E0583B
```

Figure 6.7: PRT-Attestation results as shown by the verifier. The output consists of the prover response after initiation of an attestation challenge. This response reflects the system measurements of an unmodified executable.



```
ecestudent@CB480-EE10326:~/Documents/research/checksum_gen$ ./checksum_gen /home
/ecestudent/Documents/research/ecen330/EE330_Fall2015_Student_SW/Consolidated_33
0_SW/Debug/Consolidated_330_SW.elf -s 0xE4E0583B
----------------------------------------
SWATT Mode Enabled
.text section size (bytes)            0x0002823c
Golden checksum                       0x5d6fbf53
```

Figure 6.8: Checksum-Gen for PRT-Checksum with seed provided by the security monitor as input.

Remember that in PRT-Checksum, it is the verifier's responsibility to perform checksum comparison. To do this, it must first compute the golden checksum based on the seed contained in the prover response (0xE4E0583B in this case). We use this seed as input to our Checksum-Gen utility and view the resulting checksum in Figure 6.8. Indeed the checksum as provided by the security monitor (0x5D6FBF5D) in Figure 6.7 matches the computed golden checksum.

Next we ensure that PRT-Checksum is capable of detecting a modified executable. In the same manner as done in Section 6.1.2, we modify the executable by altering one line of code. We load and run the modified executable and initiate an attestation challenge. The resulting response is shown in Figure 6.9. We run Checksum-Gen with the seed from the response, and find our golden checksum to be 0xc55f9b3c. The golden checksum does not match the one received from the security monitor, therefore we assume device compromise. Note that the attestation status indicates "PASS"; this is because in PRT-checksum, the attestation status only provides us with information regarding the PC value (since the security monitor cannot verify the checksum).

The next test is to ensure that code injected into an unused portion of memory can be detected. We set up this test in the same manner as Section 6.1.2 and receive the output shown in Figure 6.10. First, notice that the attestation status is "FAIL", thus indicating a failed PC asser-

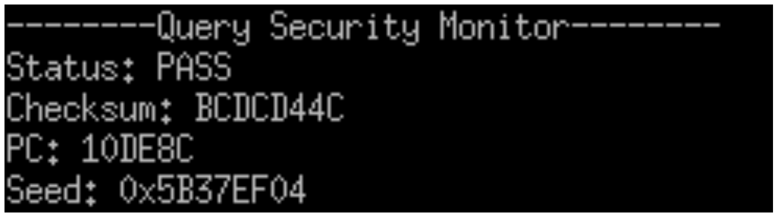

```
--------Query Security Monitor--------
Status: PASS
Checksum: BCDCD44C
PC: 10DE8C
Seed: 0x5B37EF04
```

Figure 6.9: PRT-Attestation results as shown by the verifier. The output consists of the prover response after initiation of an attestation challenge. This response reflects the system measurements of a modified executable.

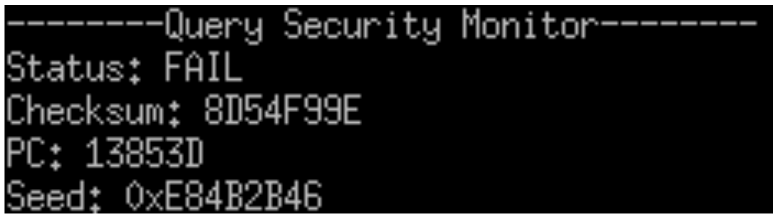

```
--------Query Security Monitor--------
Status: FAIL
Checksum: 8D54F99E
PC: 13853D
Seed: 0xE84B2B46
```

Figure 6.10: PRT-Attestation verifier output consisting of the prover response after initiation of an attestation challenge. This response reflects the system measurements of a system with extraneous code placed in unused memory.

tion. Manual examination of the PC value from the prover response reveals that a PC value of 0x0013853D was detected by the security monitor, which is not a member of $\mathbf{PC_{legal}}$.

Finally, we test that PRT-Checksum is capable of detecting an added if-statement in the attestation code. We add the if-statement from listing 6.2 to the checksum-generating code of the executable, then load and run it. The attestation returned a “FAIL” status as expected, thus demonstrating the effectiveness of PRT-Checksum in detecting added if-statements.

To test for false-positives, we load the program with the added if-statement and perform 1000 attestation challenges from the verifier serial console, each of which resulted in a “FAIL” attestation status. We then test for false-negatives by loading the original executable into device memory and performing another 1000 attestations. In each case, the checksum received matched the golden checksum, and attestation status was “PASS”.

We therefore conclude that our chosen value of R_{max} provides high resiliency to both false positives and negatives. It is worth noting however, that smaller programs will have a smaller gap between A_{min} or O_{max} due to a smaller quantity of if-statements being performed in compromised code. Therefore we speculate that a smaller executable may result in a higher probability of false positives or negatives.

6.2 Performance Results

In this section, we measure the performance impact of our attestation protocol on the test application. After all, system designers will likely be hesitant to add security features that significantly degrade their system performance.

We establish a test baseline by running the test program of 10 automated tic-tac-toe games with the security monitor disabled. Each test (set of 10 games) is seeded with the same value to maintain consistency across tests. We found that 10 games takes 4.56×10^9 processor cycles.

To measure the performance impact of the attestation mechanism, we enable the security monitor and measure the test program cycle count with various PIT intervals. For example, if the PIT timer period is set to 10 milliseconds, the security monitor will perform system measurements every 10 milliseconds. Figure 6.11 shows the impact of the security monitor in terms of total clock cycles required to run the test program to completion.

Interestingly, PRT-Checksum does not add significantly more overhead than DMA-Checksum as long as the security monitor period is greater than approximately 500 milliseconds. DMA-Checksum results in a steady .001% overhead, while PRT-Checksum hovers at approximately 1% until the period decreases below about 500 milliseconds. Periods less than 500 milliseconds begin to result in exponentially increasing overhead. In Figure 6.12 we relate these numbers to the baseline by computing the percentage overhead. The percentage overhead represents the total percentage of clock cycles that are lost to the application because of attestation.

Next we use the built-in Xilinx-SDK profiler to observe which parts of the program are utilizing the processor. Figure 6.13 shows the results. When we attempted to profile processor usage for DMA-checksum, the profiler did not have the granularity to detect the PC probe ISRs. Also, the Xilinx-SDK profiler uses interrupts to sample the software state, therefore, if the profiler interrupt priority is lower than that of the PC-probe interrupt, then the PC-probe ISR may not get profiled. However, with the results from the percent overhead measurements, we can still conclude that the percentage of processor time used in DMA-checksum is negligible.

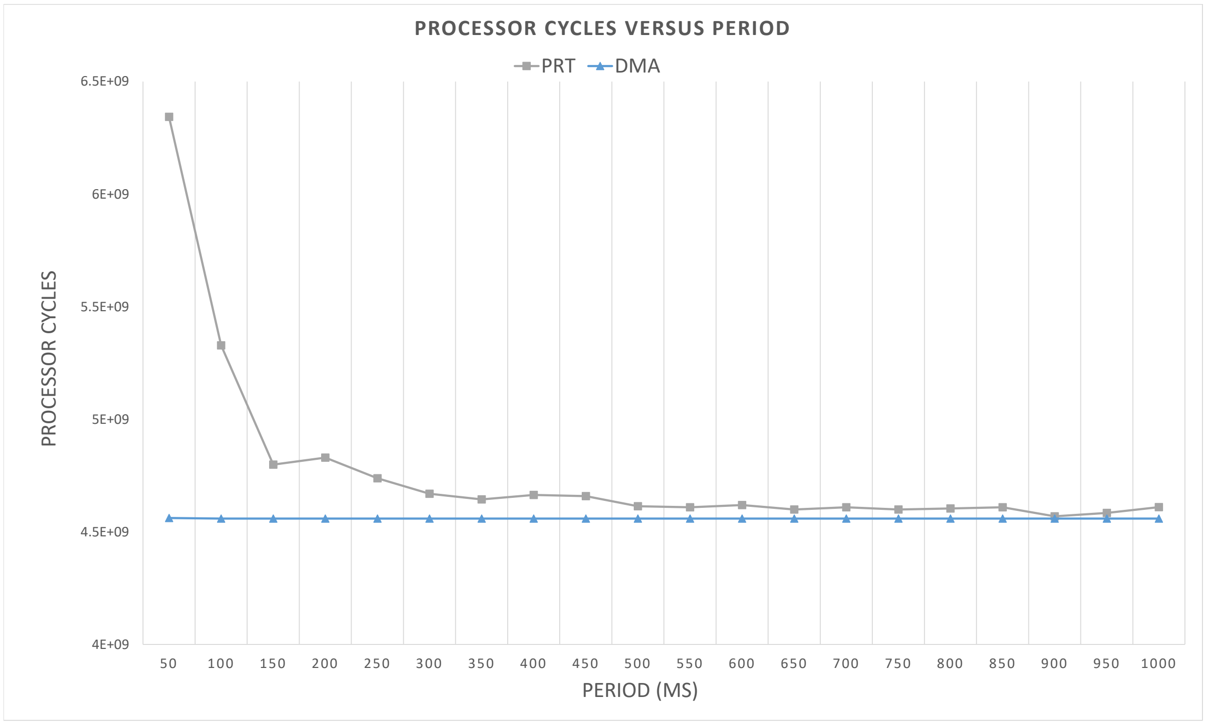


Figure 6.11: Clock cycles versus security monitor period.

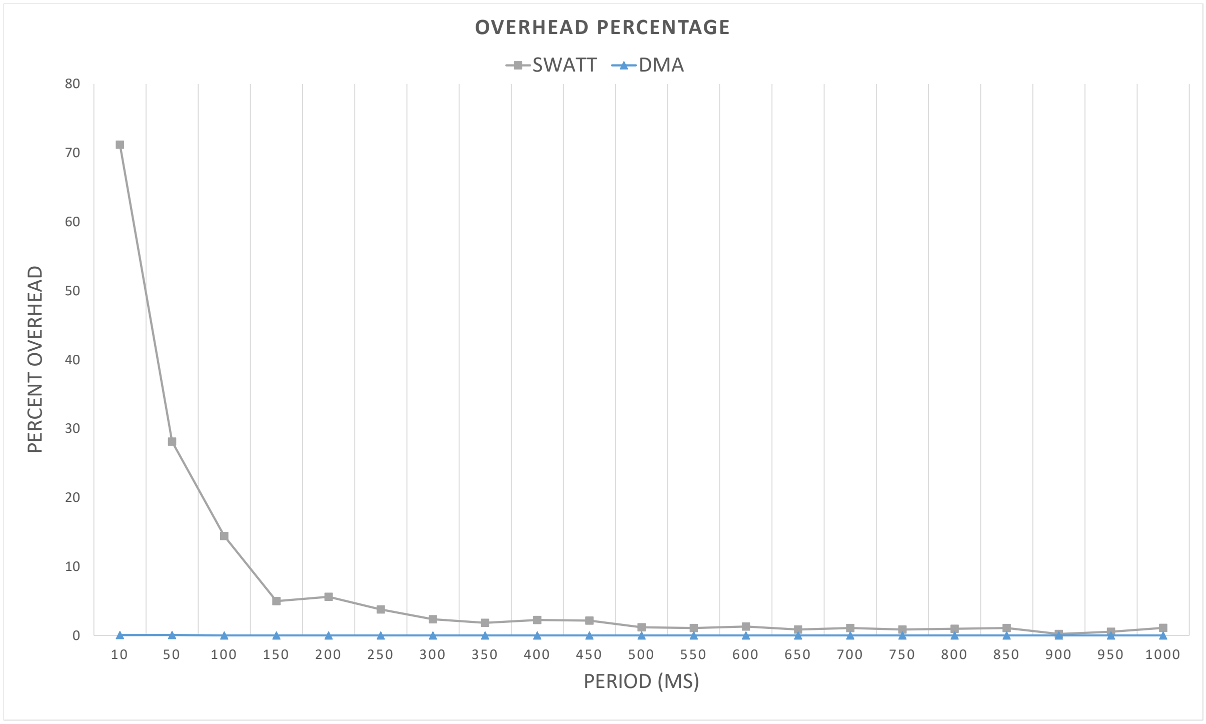


Figure 6.12: Percentage of total clock cycles lost for attestation.

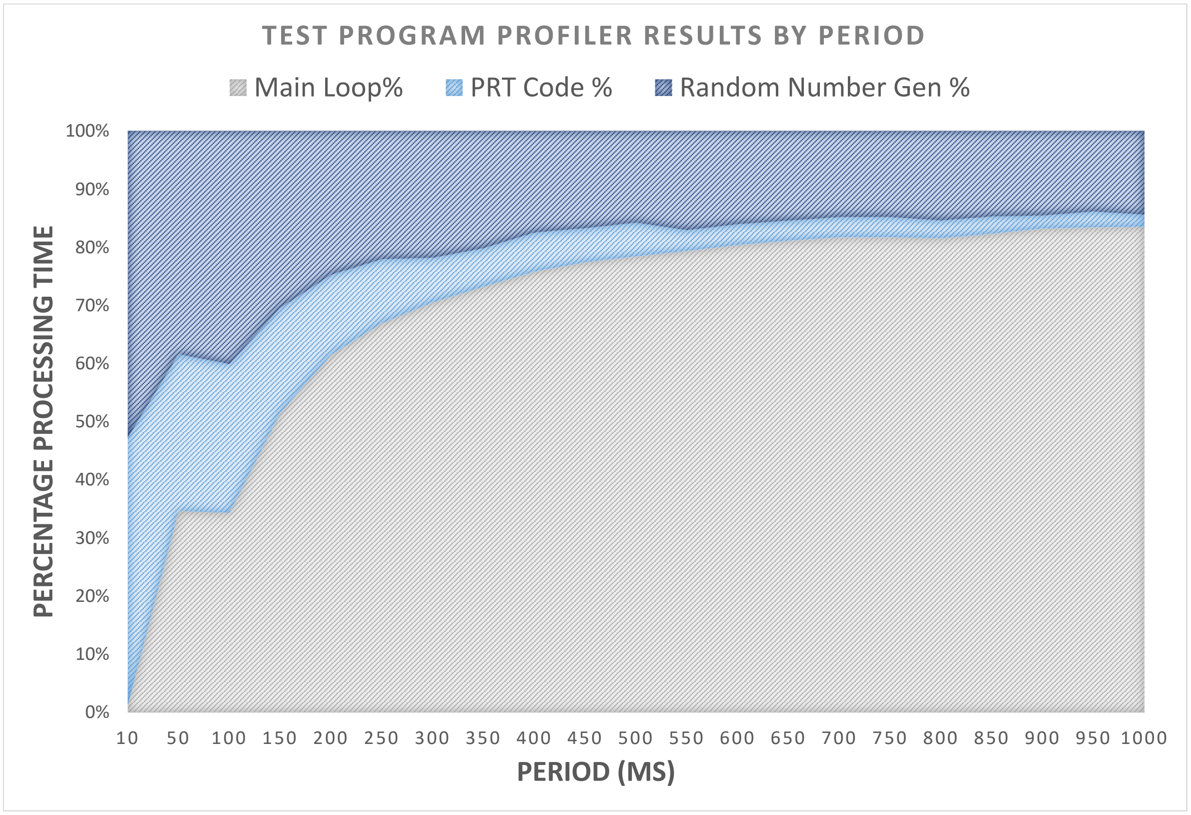


Figure 6.13: PRT-Attestation Processor Utilization.

One drawback of our method for probing the PC is that extraneous code will only be detected as long as it is *currently* running when a system measurement takes place. For example, if malicious code is placed in some unused portion of memory, it will be detected only if we happen to probe the PC while the malicious code is being executed. Imagine a compromised application where approximately 10% of the CPU time is spent in malicious code. On average, it would take 10 PC probes to detect the malicious code.

We perform an experimental test where we reduce the number of addresses in $\mathbf{PC_{legal}}$ by 10%. In other words, one-tenth of the program addresses are now deemed illegal. We do this to simulate a program that spends a small fraction of its execution time in "malicious" code. We then compute the average time required to detect an illegal PC value over 50 runs of our test program. As a reminder, the test program plays 10 complete games of tic-tac-toe. The results are shown in Figure 6.14. We also plot the expected average time to detect an illegal PC value.

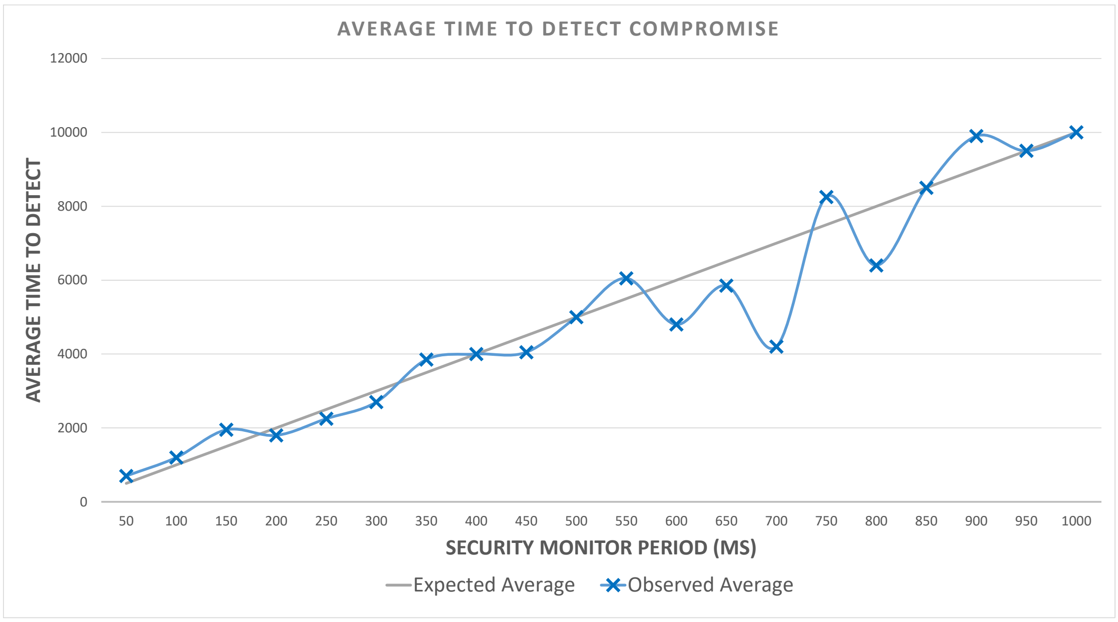


Figure 6.14: Average Time To Detect Illegal PC versus security monitor period.

There is a possibility that the malicious code and security monitor period could become synchronized in such a way that the malicious code never gets detected. In other words, there is a chance that the PC probe *always* occurs while the malicious code is not running. To help mitigate this concern, the security monitor period could be set to a pseudo-random value after each measurement is made.

6.3 Discussion

From the functional and performance results, we demonstrated that our attestation framework can be applied to various types of hardware systems. From our results, DMA-Checksum is clearly advantageous over PRT-Checksum. The increased overhead and additional complexity due to the tight timing requirements in PRT-Checksum make it inferior to DMA-Checksum, but sufficient (given proper security monitor period selection) for systems without DMA capability. Ultimately, we have demonstrated the flexibility and feasibility of our framework.

6.4 Analysis

In this section we present an analysis of the attestation framework. First, we present a protocol analysis with regard to the security properties discussed in Section 2.3. We then present a security analysis with respect to the threat model defined in Chapter 4. Finally, we discuss benefits and limitations of the proposed attestation framework.

6.4.1 Protocol Analysis

1. Measurement Diversity. We have shown how measurement diversity can be achieved through both a checksum of executable code and PC monitoring. Because the secure domain is implemented in programmable logic, a system designer could easily extend the framework to include additional measurements specific to the platform. For example, some embedded devices contain memory mapped configuration registers that are accessed through reads and writes to specific memory addresses. In a similar fashion to the memory checksum, one could create a checksum of configuration register contents.

2. Domain Separation. Our attestation protocol achieves strong domain separation by implementing the secure domain in FPGA hardware. It is possible that a software vulnerability could allow FPGA bitstream modifications, but as mentioned previously, bitstream security is beyond the scope of this thesis.

3. Self-Protection. The secure domain acts as a root of trust because it is implemented in hardware. Therefore we can assume that it boots up into a secure state. We believe that our implementation is small enough that it could possibly be formally verified, but this is left for future work.

4. Exclusive Key Access. Cryptographic keys are not externally accessible to anything outside the security monitor. The application domain is physically barred from key access.

5. Immutability. In DMA-Attestation, the protocol is immutable because attestation functionality resides purely in hardware. PRT-Attestation on the other hand, does contain some mutable components, but is designed in such a way that the secure domain can detect tampering of these components.

6. Controlled Invocation. The protocol described in Section 5.2.7 prevents unauthorized parties from initiating attestation by requiring a password and valid session token. Using sufficiently strong encryption and PRNG algorithms make it computationally infeasible for an adversary to forge an attestation request.

6.4.2 Security Analysis

1. Software Vulnerabilities. Even with careful security design practices, it is impossible to predict how an attacker might attempt to abuse and exploit a system. The attestation protocol presented here does not prevent software vulnerabilities, but it does provide a means whereby we can remotely detect compromise.

2. Denial of Service. Denial of service attacks are a major security issue and continue to grow in prevalence. We believe that the remote attestation protocol presented here could help combat denial of service attacks by providing a means to detect botnet malware, such as Mirai, that often goes unnoticed on a device.

3. Buffer Overflow. The attestation protocol presented here does not prevent buffer overflows, but as previously mentioned, it provides a reliable protocol to detect adversarial tampering. However, it is possible that remote attestation could serve as a deterrent to attempted exploits.

4. Replay Attacks. Section 5.2.7 thoroughly describes how replay attacks are prevented in our attestation framework.

5. Return Oriented Programming. Unfortunately, the attestation framework presented here does not protect against ROP techniques. This is because ROP takes advantage of code that already exists on the system without modifying the executable. Additionally, the PC will be executing instructions within the legal range, so this approach cannot detect ROP exploits.

6.4.3 Benefits

Flexibility

One of the major advantages of the attestation framework presented here is that it can easily be updated or adapted to suit various types of systems. We have already demonstrated how PRT-Checksum can be used for systems without DMA, and DMA-Checksum for those with DMA capability. Security is largely a “cat and mouse” game, meaning that the idea of security is constantly changing and adapting. A “secure” system may suddenly be made insecure by the discovery of a new vulnerability, at which point the system designers may be able to make it secure again by patching the vulnerability. The ability to update the security monitor and other secure domain components is extremely advantageous. If a previously unknown vulnerability is discovered, the FPGA can be updated to close the vulnerability. However, the FPGA update process itself may present new security vulnerabilities, but this is beyond the scope of this thesis. Secure remote update mechanisms are an entirely different area of research.

Cryptographic algorithms have a finite lifetime. As computational power increases and becomes cheaper, cryptographic algorithms must be updated and strengthened to resist compromise. If current cryptographic algorithms become deprecated and new ones replace them, it would be relatively easy to update the attestation framework to utilize the latest algorithms. This could perhaps even be achieved remotely, through secure remote update processes.

Low Overhead

We have shown that DMA-Checksum attestation incurs a very small performance penalty on a system (.001%), and below a certain execution frequency, PRT-Checksum also incurs low system overhead. In an ideal system with DMA capability, the security monitor operates with minimal processor intervention and therefore essentially negligible overhead.

Fast Verifier-Prover Response Time

In essence, attestation is continually taking place in our attestation protocol. The security monitor periodically makes system measurements and determines security status. Therefore, when a verifier sends an attestation challenge, it is essentially just querying the current security status within the security monitor. This means that the verifier does not have to wait for the prover to compute a proof, as is common in many attestation protocols. This is especially beneficial for large programs, where (as demonstrated in Section 6.1.1) the time to generate a memory checksum is non-trivial. This is especially important for user-facing applications where response time is crucial to user experience.

6.4.4 Limitations

Cost

The major drawback to this attestation framework is the added cost of the FPGA. However, SoCs integrating FPGAs with a processor are becoming more commonplace. As the cost of FPGAs decline, we believe that FPGA cost will become less of an issue in the near future.

Power

A major consideration in embedded systems design is power draw. Mobile phones and other devices that run on battery power dominate our daily lives. Obviously, battery powered devices must be designed so as to maximize battery life. Our attestation framework may not be well suited to battery powered devices, but a thorough power study would be required to make any claims in this regard.

The addition of an FPGA to an embedded system will inevitably consume more power than one without. However, low-power FPGAs do exist and FPGA designs are continually improving and becoming more power efficient.

Bitstream Security

We briefly mentioned the issue of bitstream security in Section 5.2.1. For this attestation protocol to be secure, it is imperative that the bitstream be immutable. If an adversary is able to upload or modify the bitstream, therefore modifying the secure domain, then we can no longer make secure claims about our system. Methods to ensure trust in FPGA bitstreams have been explored and are an active area of research [17], [49], [50].

Single Process Systems

Currently, this protocol is only suitable for single process systems with a single thread of execution. Therefore, a system requiring an operating system (OS) would not be well suited for this type of attestation. An OS will determine where to load an executable into memory (rather than a linker script); therefore the task of calculating $\mathbf{PC}_{\mathbf{legal}}$ becomes problematic. It could still be possible to perform DMA-Checksum on a device running an OS, but the security monitor would need to have information regarding the executable location in memory.

CHAPTER 7. CONCLUSION

Cyber-crime is at an all time high, and will likely become more commonplace. More and more devices are being built with Internet connectivity, opening them up to cyber attacks. These devices are often designed without security in mind, leaving them vulnerable to attack. The work in this thesis builds upon previously proposed attestation techniques, namely SWATT and FPGA-Based Remote Code Integrity Verification, to provide a flexible framework for remote attestation of Internet connected embedded devices.

7.0.1 Contributions

The main contributions of this thesis are summarized as follows:

- A flexible remote attestation framework that is adaptable and extensible.
- A proof-of-concept implementation of the framework to demonstrate feasibility. The implementation successfully detects tampered executables and unauthorized code stored in memory.
- Demonstration of adaptability to systems with different hardware features.
- A study of the performance impact of various security monitor periods.
- A survey of embedded systems security and proposed attestation techniques.

7.0.2 Future Work

As mentioned in the previous chapter, our framework is vulnerable to ROP techniques. An area of future work would be to build in resistance to ROP, perhaps through implementations of existing ROP defense mechanisms. Also, because the security monitor frequently probes the PC, it may be possible to construct some type of histogram or control tree, representing the historical execution path. Using this data, the security monitor could possibly predict if control flow seems unusual, hence detecting ROP exploit. We are unsure if this is feasible in FPGA hardware, however, or if a secure co-processor would be required for such a task.

Another interesting area not explored in this paper would be the power impact of the protocol. We have demonstrated correct functionality and feasibility of the protocol, but if power requirements are too steep, it may not be of much utility to embedded systems designers wishing to optimize for battery life.

With the growing popularity of operating systems such as Linux, the attestation framework presented here would be even more useful if it could be adapted to work with multi-process operating systems. This would require the security monitor to have some knowledge regarding the OS memory management, so as to allow the security monitor to construct checksums of the correct memory regions.

Currently, many of the security monitor parameters (such as shared symmetric keys, $\mathbf{PC_{legal}}$, etc.) must be hard-programmed into the FPGA bitstream. A configuration protocol that would allow a verifier to update security monitor parameters could be extremely beneficial to the framework. Additionally, a key exchange protocol such as Diffie-Hellman would be useful for updating the shared keys in a secure manner.

REFERENCES

[1] P. E. Ross, “Hackers Commandeer a Moving Jeep - IEEE Spectrum,” p. 1, 2015. [Online]. Available: http://spectrum.ieee.org/cars-that-think/transportation/self-driving/hackers-take-control-of-a-moving-jeep 1

[2] A. Drozhzhin, “Black Hat USA 2015: The full story of how that Jeep was hacked,” 2015. [Online]. Available: https://www.kaspersky.com/blog/blackhat-jeep-cherokee-hack-explained/9493/ 1

[3] Gartner Research, “Gartner Says 8.4 Billion Connected ”Things” Will Be in Use in 2017, Up 31 Percent From 2016,” 2017. [Online]. Available: https://www.gartner.com/newsroom/id/3598917 1

[4] J. Radcliffe, “Hacking Medical Devices for Fun and Insulin: Breaking the Human SCADA System,” in *Black Hat Conference presentation slides*, 2011. 1

[5] N. Allen, “Cybersecurity weaknesses threaten to make smart cities more costly and dangerous than their analog predecessors,” *USApp–American Politics and Policy Blog*, 2016. 1

[6] A. Stavrou, J. Voas, I. Fellow, C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, “DDoS in the IoT: Mirai and Other Botnets,” *Computer*, vol. 50, no. 7, pp. 80–84, 2017. 1

[7] M. Patton, E. Gross, R. Chinn, S. Forbis, L. Walker, and H. Chen, “Uninvited connections: A study of vulnerable devices on the internet of things (IoT),” *Proceedings - 2014 IEEE Joint Intelligence and Security Informatics Conference, JISIC 2014*, pp. 232–235, 2014. 1

[8] Z. K. Zhang, M. C. Y. Cho, C. W. Wang, C. W. Hsu, C. K. Chen, and S. Shieh, “IoT security: Ongoing challenges and research opportunities,” *Proceedings - IEEE 7th International Conference on Service-Oriented Computing and Applications, SOCA 2014*, pp. 230–234, 2014. 1, 20

[9] B. A. Kuperman, C. E. Brodley, H. Ozdoganoglu, T. N. Vijaykumar, and A. Jalote, “Detection and Prevention of Stack Buffer Overflow Attacks,” *Communications of the ACM*, vol. 48, no. 11, pp. 50–56, nov 2005. 1

[10] C. Cowan, F. Wagle, Calton Pu, S. Beattie, and J. Walpole, “Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade,” in *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX’00*, vol. 2. IEEE Comput. Soc, 2003, pp. 119–129. 1

[11] R. Langner, “Stuxnet: Dissecting a Cyberwarfare Weapon,” *IEEE Security and Privacy*, vol. 9, no. 3, pp. 49–51, 2011. 2

[12] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik, “A Minimalist Approach to Remote Attestation,” *Proceedings of the Conference on Design, Automation & Test in Europe*, no. 244, pp. 1–6, 2014. 2, 7

[13] A. Seshadri, A. Perrig, L. Van Doom, and P. Khosla, “SWATT: SoftWare-based ATTestation for embedded devices,” *Proceedings - IEEE Symposium on Security and Privacy*, vol. 2004, pp. 272–282, 2004. 2, 16, 31

[14] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla, “SCUBA: Secure Code Update By Attestation in sensor networks,” *WiSe ’06: Proceedings of the 5th ACM workshop on Wireless security*, pp. 85–94, 2006. 2

[15] N. Agarwal and K. Paul, “XEBRA: XEn Based Remote Attestation,” in *IEEE Region 10 Annual International Conference, Proceedings/TENCON.* IEEE, nov 2017, pp. 2383–2386. 2, 11, 15, 35

[16] P. Koeberl, S. Schulz, A. Sadeghi, and V. Varadharajan, “TrustLite: A Security Architecture for Tiny Embedded Devices,” *Proceedings of the European Conference on Computer Systems (EuroSys)*, pp. 1–14, 2014. 2

[17] C. Basile, S. D. Di Carlo, and A. Scionti, “FPGA-based remote-code integrity verification of programs in distributed embedded systems,” *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews*, vol. 42, no. 2, pp. 187–200, 2012. 2, 18, 35, 68

[18] K. Eldefrawy, A. A. Francillon, D. Perito, G. Tsudik, K. E. Defrawy, A. A. Francillon, D. Perito, and G. Tsudik, “SMART: Secure and Minimal Architecture for (Establishing a Dynamic Root of Trust,” in *NDSS 2012, 19th Annual Network and Distributed System Security Symposium, February 5-8, San Diego, USA*, vol. 12, San Diego, UNITED STATES, 2012, pp. 1–15. 2, 18

[19] D. Fu and X. Peng, “TPM-based remote attestation for Wireless Sensor Networks,” *Tsinghua Science and Technology*, vol. 21, no. 3, pp. 312–321, 2016. 2

[20] A. Mey and S. Hoff, “Nearly Half of All U.S. Electricity Customers Have Smart Meters,” 2017. [Online]. Available: https://www.eia.gov/todayinenergy/detail.php?id=34012 5

[21] X. Fang, S. Misra, G. Xue, and D. Yang, “Smart Grid The New and Improved Power Grid: A Survey,” *IEEE Communications Surveys & Tutorials*, vol. 14, no. 4, pp. 944–980, 2012. 5

[22] B. Krebs, “FBI: Smart Meter Hacks Likely to Spread,” 2012. [Online]. Available: https://krebsonsecurity.com/2012/04/fbi-smart-meter-hacks-likely-to-spread/ 5

[23] C. W. Ten, C. C. Liu, and G. Manimaran, “Vulnerability Assessment of Cybersecurity for SCADA Systems,” *IEEE Transactions on Power Systems*, vol. 23, no. 4, pp. 1836–1846, 2008. 7

[24] Y. Chahid, M. Benabdellah, and A. Azizi, “Internet of things security,” *2017 International Conference on Wireless Technologies, Embedded and Intelligent Systems, WITS 2017*, 2017. 7

[25] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of remote attestation," *International Journal of Information Security*, vol. 10, no. 2, pp. 63–81, 2011. 7

[26] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996. 8

[27] C. Moratelli, S. Johann, M. Neves, and F. Hessel, "Embedded virtualization for the design of secure IoT applications," *Proceedings of the 27th International Symposium on Rapid System Prototyping Shortening the Path from Specification to Prototype - RSP '16*, pp. 2–6, 2016. 11

[28] R. Kaiser and S. Wagner, "Evolution of the PikeOS microkernel," *First International Workshop on Microkernels for Embedded Systems*, no. January, 2007. 11

[29] G. Heiser and B. Leslie, "The OKL4 Microvisor: Convergence point of microkernels and hypervisors," *Proceedings of the first ACM asia-pacific workshop . . .*, pp. 19–23, 2010. 11

[30] A. Patel, M. Daftedar, M. Shalan, and M. W. El-Kharashi, "Embedded hypervisor xvisor: A comparative analysis," *Proceedings - 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015*, pp. 682–691, 2015. 11

[31] A. Celesti, D. Mulfari, M. Fazio, M. Villari, and A. Puliafito, "Exploring Container Virtualization in IoT Clouds," *2016 IEEE International Conference on Smart Computing, SMARTCOMP 2016*, no. Lxc, 2016. 11

[32] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 171–172, 2015. 11

[33] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: A performance comparison," *Proceedings - 2015 IEEE International Conference on Cloud Engineering, IC2E 2015*, pp. 386–393, 2015. 11

[34] N. L. Petroni, T. Fraser, J. Molina, and W. a. Arbaugh, "Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor," *USENIX Security'04*, pp. 179–194, 2004. 11

[35] Trusted Computing Group Incorporated, "TCG Specification Architecture Overview." [Online]. Available: https://www.trustedcomputinggroup.org/wp-content/uploads/TCG{_}1{_}4{_}Architecture{_}Overview.pdf 11

[36] H. Krawczyk, R. Canetti, and M. Bellare, "HMAC: Keyed-hashing for message authentication," pp. 1–11, 1997. 12

[37] N. Leveson and C. Turner, "An investigation of the Therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, 1993. 20

[38] N. Falliere, L. Murchu, and E. Chien, "W32. stuxnet dossier," *Symantec Security Response*, vol. 14, no. February, pp. 1–69, 2011. [Online]. Available: http://large.stanford.edu/courses/2011/ph241/grayson2/docs/w32{_}stuxnet{_}dossier.pdf 20

[39] A. Costin, J. Zaddach, and A. Francillon, “A Large Scale Analysis of the Security of Embedded Firmwares,” *USENIX Security*, 2014. 21

[40] N. D. Matsakis and F. S. Klock, “The Rust Language,” *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103–104, 2014. 21

[41] Z. Li and Q. Liao, “Toward a Monopoly Botnet Market,” *Information Security Journal*, vol. 23, no. 4-6, pp. 159–171, 2014. 21

[42] P. Syverson, “A Taxonomy of Replay Attacks,” *Proceedings The Computer Security Foundations Workshop VII*, pp. 187–191, 1994. 25

[43] M. Prandini and M. Ramilli, “Return-Oriented Programming,” *IEEE Security & Privacy*, vol. 10, no. 6, pp. 84–87, nov 2012. 25

[44] H. Shacham, “The Gemoetry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86),” *Proceedings of the 14th ACM conference on Computer and communications security - CCS ’07*, p. 552, 2007. 28

[45] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, “When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC,” *ACM Transactions on Information and System Security*, vol. 15, no. 1, pp. 1–34, 2012. 28

[46] JonathanSalwan, “ROPgadget.” [Online]. Available: https://github.com/JonathanSalwan/ROPgadget 28

[47] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, “G-Free: Defeating Return-Oriented Programming Through Gadget-less Binaries,” in *Proceedings of the 26th Annual Computer Security Applications Conference on - ACSAC ’10*. New York, New York, USA: ACM Press, 2010, p. 49. 28

[48] L. Davi, A. Sadeghi, and M. Winandy, “ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks,” *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security - ASIACCS ’11*, p. 40, 2011. 28

[49] S. Trimberger, “Trusted design in FPGAs,” in *Proceedings of the 44th annual conference on Design automation - DAC ’07*, vol. 9781441980. New York, New York, USA: ACM Press, 2007, p. 5. 35, 68

[50] M. M. Parelkar and K. Gaj, “Implementation of EAX mode of operation for FPGA bitstream encryption and authentication,” *Proceedings - 2005 IEEE International Conference on Field Programmable Technology*, vol. 2005, pp. 335–336, 2005. 35, 68

[51] Digilent Incorporated, “Zybo Reference Manual.” [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/zybo/reference-manual 49

[52] Xilinx Incorporated, “Zynq-7000 All Programmable SoC Technical Reference Manual.” [Online]. Available: https://www.xilinx.com/support/documentation/user{_}guides/ug585-Zynq-7000-TRM.pdf 49

[53] H. Hsing, “Tiny AES.” [Online]. Available: https://opencores.org/project,tiny{_}aes, Overview 50