Design and Implementation of PUF Based Protocols for Remote Integrity Verification

Shravya Gaddam

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Patrick R. Schaumont, Chair
Leyla Nazhandali
Lynn Abbott

June 27, 2016
Blacksburg, Virginia

Keywords: Physical Unclonable Functions, ECDSA,
Elliptic Curve Cryptography, Fuzzy Extractors, Strong Extractors

Design and Implementation of PUF Based Protocols for Remote Integrity Verification

Shravya Gaddam

(ABSTRACT)

In recent years, there has been a drastic increase in the prevalence of counterfeiting within the electronics supply chain. At the same time, high-end commercial off-the-shelf components like FPGAs and expensive peripherals are making their way onto printed circuit boards. Manufacturers of such PCBs lose billions of dollars as well as their reputation when counterfeiting incidents are revealed within their supply chain. Moreover, there are several safety and security implications of using PCBs with counterfeit components. In this context, it is useful to enable remote integrity checking of these PCBs to identify and mitigate any safety or security concerns when they are deployed. Typical integrity checks look for the presence of an identifier embedded within a secure memory on the PCB. This approach is now being replaced by hardware intrinsic identifiers based on Physical Unclonable Functions or PUFs. Such identifiers can be used to establish trust within any component on a PCB.

In this thesis, we present two PUF based protocols for remote integrity checking of PCBs by Manufacturers or end users. We propose one of the protocols for a special case of remote integrity checking - the Third Party Verification. The protocols are demonstrated using prototypes running on two different platforms - Altera DE2-115 and TI MSP430. Finally, we evaluate their performance on these prototypes and determine the feasibility of their use.

~ To my family, for all their love and support ~

Acknowledgments

I would like to thank my advisor, Dr.Patrick Schaumont, for his support and guidance during my Masters. Being a part of the Secure Embedded Systems group under his advisement has been an amazing learning experience - one that I am deeply grateful for. I would like to thank Dr.Leyla Nazhandali and Dr.Lynn Abbott for serving on my thesis committee. I am grateful to Dr. Paul Plassmann for his support towards my thesis defense.

I was able to finish my thesis due to the love and encouragement of my family. Even though they were not by my side during my Masters, their support has helped me greatly.

I would like to express my gratitude to my labmate Aydin, for his insights and guidance with my research. I am thankful to Harsha and Carol for their contribution to this thesis.

This work was supported in part by CISCO Systems Inc, and by National Science Foundation grant no 1314598.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Preliminaries

The electronics industry faces billions of dollars in losses each year due to counterfeit components [19]. When these non-standard parts make their way onto critical systems, safety and reliability are also compromised [7]. It is useful to enable a remote integrity check on such systems so that manufacturers or end users can verify that the components on their devices are authentic. In this thesis, we provide a protocol for third party verification of device integrity. We also demonstrate the operation of two remote verification protocols on two platforms - TI MSP430 and Altera DE2-115.

This chapter introduces the problem of counterfeiting and remote integrity verification. We discuss the threat model and security objectives for such an integrity check. Some preliminary knowledge on Physical Unclonable Functions(PUFs) and integrity checking using PUFs is also provided. Finally, we present existing solutions for remote integrity verification and our contributions. In this thesis, we use the term component to refer to individual ICs like SRAM and a device is the end product of the electronics supply chain and contains atleast one component.

1.1 Counterfeiting and Remote Integrity Verification

Counterfeiting can be defined as the act of fraudulently manufacturing, altering, distributing, or offering a product or package that is represented as genuine [7]. The counterfeit components are passed down to the consumer through the electronics supply chain as shown in Figure 1.1. Such components may be used in devices with a wide variety of applications from consumer electronics to aerospace, health and defense systems. According to [19], the potential risk to the electronics industry due to counterfeit components in 2011 is 169 billion dollars. Apart from the monetary losses, such devices also damage the manufacturer's reputation, cause critical system failure and result in health, safety, security issues. Due to the prevalence of this problem, counterfeiting has become a major concern for the electronics industry.

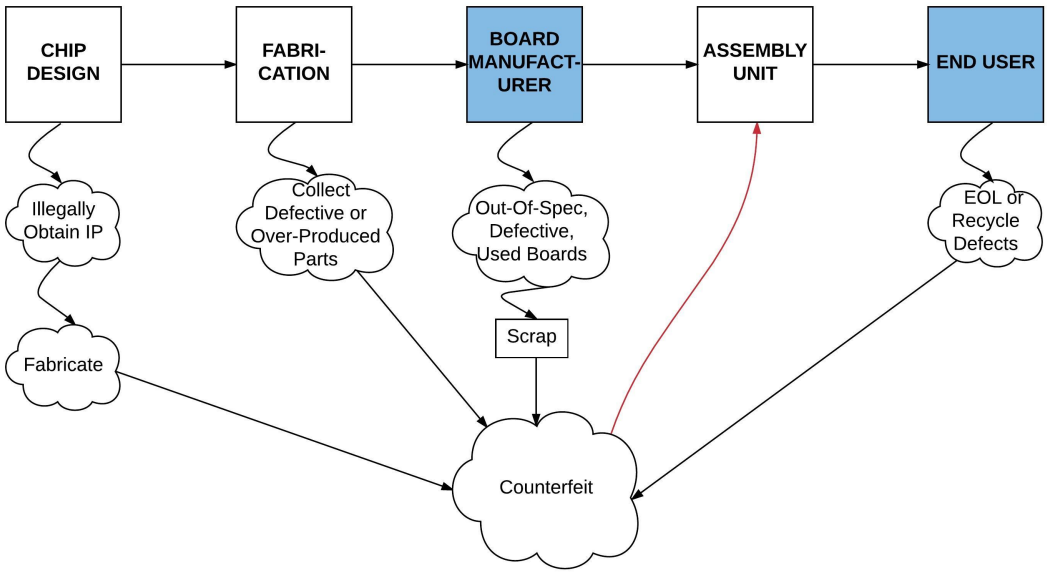


Figure 1.1: Simplified Diagram showing the sources of Counterfeit Components in the Electronics Supply Chain

It is important for Manufacturers of electronic devices to ensure that the components on the PCBs remain authentic when they reach the consumer through the supply chain. This can

be achieved by designing an integrity check to verify that all components on the consumer device are genuine. This check must reveal hardware tampering, or tamper within the supply chain from manufacturing/deployment to remote installation. The Manufacturer should be able to run this check on a consumer device from a server installed in a remote location. Also, it should allow for verification of a large number of devices i.e. it should be scalable. In this thesis, we provide two protocols for remote integrity verification, each for a different use case.

1.2 Threat Model

In a typical deployment scenario, Manufacturers communicate with their devices over the network to perform upgrades, check for system failure etc. With such an infrastructure already in place, it is easier to use the same network to communicate with the device for integrity checks. This means that the communication channel between the Manufacturer and the device may not be secure. An adversary may gain control over this communication channel and try to retrieve any secret protocol data. The integrity check must be designed keeping this in mind. We expect the integrity checking protocol to operate under the following threat model as shown in Figure 1.2.

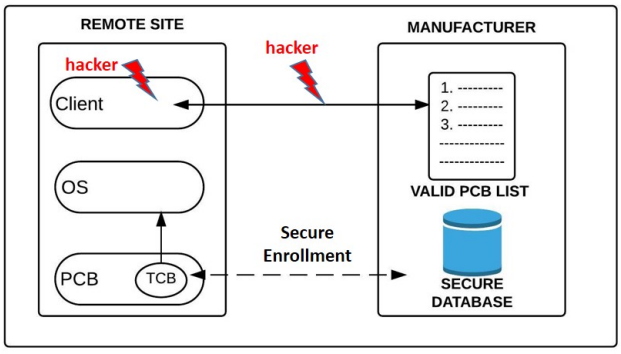


Figure 1.2: Adversary Threat Model

It is safe to assume that the communication link between the device and the verifier is not secure. This makes it easy for the adversary to

- eavesdrop - Listen to messages on the communication channel
- tamper - modify messages between entities
- impersonate - send/receive messages as a valid entity

In addition to control over the communication channel, the adversary also has access to the non-volatile memory on the device. We assume that the device has a minimal trusted computing base (TCB) to enforce privileged access to the secret protocol data. The adversary may tamper with any application level software running on the device whereas the OS or at least part of the OS is protected by the TCB. We do not counter side channel or fault attacks as part of this thesis.

1.3 Existing Anti-Counterfeiting Solutions

Commercial anti-counterfeiting solutions involve programming a OTP or ROM with a unique identifier. However, this technique fails to establish trust on all components of a device.While fuses provide another means of detecting counterfeit components, burned fuses can be replaced and therefore, not a very secure approach. Hardware metering [14] offers a better solution by generating a hardware intrinsic ID based on fabrication masks of the components. With passive hardware metering, the IP Designer can track the components based on their identifiers. Active hardware metering allows the IP designer to lock, access and unlock components without sharing this information with the foundry. However, many hardware metering techniques rely on pre-synthesis modification of the component design. Our problem involves enabling PCB Manufacturers or end users to verify the integrity of their devices through the supply chain. Hardware metering is enabled by IP Designers to detect counterfeiting by untrusted foundries. Therefore, this approach is not suitable for our problem.

1.4 Physical Unclonable Functions

A straight forward approach to enable remote integrity checks is the use of an identifier stored in a secure memory on the device. However, this only ensures the presence of the secure memory and does not represent the integrity of other components. Physical Unclonable Functions(PUFs) [9] can be used to generate an identifier that truly verifies the integrity of the device.

The PUF of an object is a function of its intrinsic properties which are developed during manufacturing. It is unique to the specific instance of the object and is almost impossible to clone. This property of PUFs is useful to verify the integrity of the electronic components on a device. PUFs generally work on challenge-response authentication. The PUF is given a challenge to evaluate and its response is collected. This response is unique to the component and can be used to identify it. However, PUF responses are noisy and therefore, using PUFs in authentication protocols requires additional analysis. The quality of a PUF can be defined using two properties - uniqueness and reliability. Uniqueness is a measure of the independence of PUF responses between different instances of the same component. Reliability represents the noisiness of different responses of the same instance of a component. We can measure the uniqueness and reliability of a PUF by empirical measurement or simulation.

PUF responses from individual components can be merged to create a board level identifier. This verifies that all the components used to generate the identifier are not compromised. When merging PUF responses, we must take into account the difference in probability of bit flips both within and between devices. This is to ensure that the integrity check does not allow faulty devices (false accept) or fail authentic devices (false reject) due to noise. The technique to merge PUF responses to meet a given security level, false reject/accept constraints is described as Fusion PUF in [1]. In our implementation, we have a security level of 80 bits with an error probability of one part in a million.

In PUF-based authentication protocols, the PUF response is a secret key which is enrolled

once and later verified by means of cryptographic operations. Compared to traditional protocols, this secret key is noisy and error coding is required to generate a stable key. During manufacturing, a large number of challenge-response pairs corresponding to the device are enrolled into a secure database. At a later stage, the verifier selects a random challenge which is given as an input to the PUF on the device. The response is then compared to the PUF response in the verifier's database. The noise in the PUF response of the device should be within a pre-defined limit to pass the integrity check. Also, it is important to not repeat a challenge as the communication channel between the device and the verifier may not be secure.

Error correction techniques can be used to avoid maintaining large number of challenge-response pairs(CRPs) at the verifier. The device sends information about the errors in PUF response in an encoded format called the helper data. Using this information and an error decoding algorithm, the PUF response at the verifier can be used to reconstruct the device response. This procedure of recovering the PUF response from a previously enrolled value is called Fuzzy extraction [6].

1.4.1 Fuzzy Extraction

PUF responses are noisy and biased. In order to compare two PUF responses, we need an error correction mechanism to remove the noise. This can be achieved securely using fuzzy extractors [6]. The design of a fuzzy extractor includes a secure sketch which is a pair of Sketch and Recover algorithms as shown in the Figure 1.3.

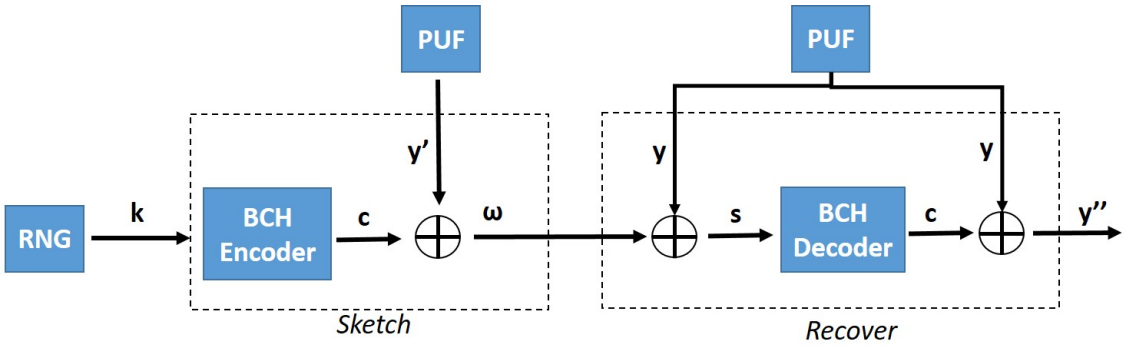


Figure 1.3: Code-offset Fuzzy Extraction

The Sketch procedure takes the PUF response(y) as an input and generates a random bit string ω called the helper data. A uniformly random number is selected and encoded using an error correction code (BCH) into a code word (c). The helper data ω can then be calculated as $\omega = y \oplus c$.

The Recover procedure takes the original PUF response y and helper data ω to recover the noisy PUF response y'. A syndrome is calculated as $s = \omega \oplus y$. The syndrome is decoded using the BCH decode algorithm and the noisy PUF response y' is reconstructed using the decoded value (e) and the original PUF response (y) using $y'' = y \oplus e$ where y" is the reconstructed PUF response. This procedure is successful as long as the number of bit errors between the original and noisy PUF response is within the error correction capacity of the BCH code.

Therefore, using Fuzzy Extraction, the verifier can maintain a much smaller database containing single PUF responses which reduces the overall maintenance cost. Also, since the verifier and the device know the same PUF response, we can now generate secret keys for cryptographic operations. A hash of appropriate length (depending on the desired security level) can be generated from the PUF response. This is called strong extraction.

1.4.2 Strong Extraction

Cryptographic keys are uniformly random bit strings with high entropy to ensure maximum security. Since PUF responses are heavily biased, we need a mechanism to extract a random string for keys. Strong extraction is a technique used to extract randomness from a biased or low-entropy input [17]. Cryptographic hash functions can be used to build strong extractors [3]. We use the SHA-256 hash function to generate keys for our proposed protocol.

Using the above concepts as building blocks, we can design protocols for remote verification with PUFs.

1.4.3 Security Objectives

The protocol must be designed with the following security objectives for the integrity check to be secure. We assume PUF enrollment to be a secure operation.

- *Correctness* - Any device that has been enrolled during manufacturing and not tampered with must pass the integrity check. Also, any device which does not produce the correct PUF response must fail the integrity check. The protocol must ensure that noisy PUF responses do not result in false rejects or false accepts.
- *Confidentiality* - Any adversary eavesdropping on the communication channel must not be able to obtain the value of the PUF response. If the PUF response is revealed, the adversary may use it to authenticate a faulty device.
- *Resistance to Replay Attacks* - The adversary should not be able to replay information from any previous protocol run to authenticate a faulty device.
- *Resistance to Impersonation Attacks* - The adversary should not be able to impersonate the verifier or the device. Otherwise, the adversary may gain access to sensitive information which is normally revealed only to a valid entity.

1.5 Contributions

This thesis brings the following contributions to protocol design using PUFs –

- We propose a protocol for third party verification of device integrity. This protocol is based on the ECDSA signature scheme which enables its use on resource constrained platforms.
- We evaluate the proposed protocol and a mutual authentication scheme on two different platforms - TI MSP430 and Altera DE2-115. We provide the results of our implementations and show that PUF based protocols are a feasible and cost effective method to extend physical trust into all components of a system.

A part of this work is described in the paper:

- A. Aysu, S. Gaddam, H. Mandadi, C. Pinto, L. Wegryn, P. Schaumont, "A Design Method for Remote Integrity Checking of Complex PCBs," Design, Automation & Test in Europe (DATE 2016), Dresden, Germany, March 2016.

1.6 Organization

The rest of this thesis describes the two protocols mentioned above and their implementation. Chapters 2 and 3 describe the protocols and their security analysis. Chapters 4 and 5 discuss about the implementation details of these protocols on two platforms - TI MSP430 and Altera DE2-115.

The individual chapters are structured as follows: Chapter 2 presents a detailed description of the Mutual Authentication scheme and an analysis of the security properties it satisfies. Chapter 3 provides preliminary knowledge about the ECDSA protocol and explains the pro-

posed third party verification protocol. We also present an analysis of the security objectives achieved by this protocol.

Chapter 4 discusses the implementation details of the two protocols on TI MSP430 and Altera DE2-115. We describe the hardware architecture of the two platforms along with the flow of control between various software blocks. Chapter 5 provides the results of our implementation and an analysis which demonstrates the feasibility of using PUFs for remote integrity checks. Finally, Chapter 6 summarizes our contributions.

1.7 Related Work

Authentication protocols based on Physical Unclonable Functions operate with two entities - a verifier who has access to a secure database of PUF responses and a prover who holds the device containing the PUF [4, 8, 13, 16]. Our proposed protocol differs from the conventional PUF based authentication by enabling secure third party verification of PUF responses. Researchers have also evaluated the feasibility of using PUFs for authenticating highly constrained platforms like RFID tags [5, 12]. However, these approaches require a large number of challenge response pairs to be maintained by the verifier which may prove to be expensive for devices that are manufactured on a large scale. In contrast, our implementation requires the verifier to maintain a single PUF response per device.

Chapter 2

Mutual Authentication Protocol

We follow the Reverse Fuzzy Extractor Protocol as defined in [15]. The two parties in this protocol are - the server at the verifier which has access to the secure database and the device that is to be verified. This topology is shown in Figure 2.1. In this section, we give a detailed description of protocol operation and an analysis of the security properties of the protocol.

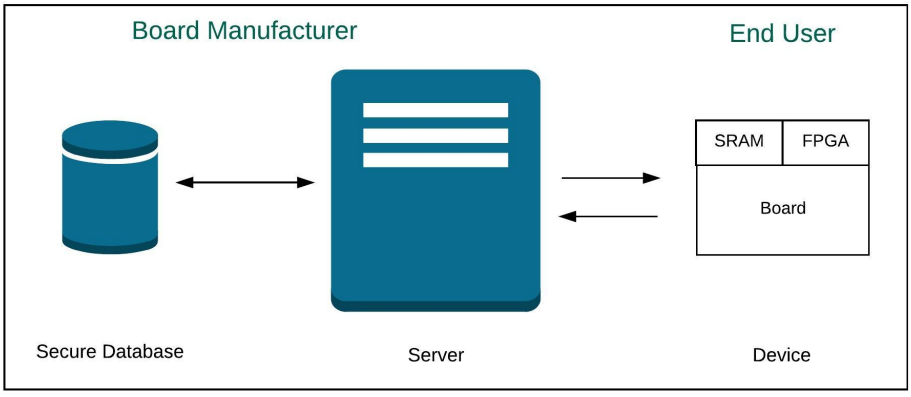


Figure 2.1: Topology for the Mutual Authentication Protocol

2.1 Notation

We use the following terms and notation while describing the protocol –

- puf_i - PUF embedded on the device
- y_i - PUF response from y_i collected during manufacturing that is enrolled into a secure database
- ID_i - manufacturer serial number of the board or a tag reference to the board in the secure database
- y_i' - noisy PUF response collected from the device during verification
- *Gen*(y_i') - Function used to generate helper data
- ω_i - helper data generated using *Gen*(y_i') to help correct bit errors in y_i
- r_1, r_2 - random nonces
- *Recover*(y_i, ω_i) - Function used to recover the noisy PUF response from the device
- y_i'' - PUF response recovered at the server
- *Hash* - Cryptographic Hash Function
- u_1, u_2 - Computed Hash values

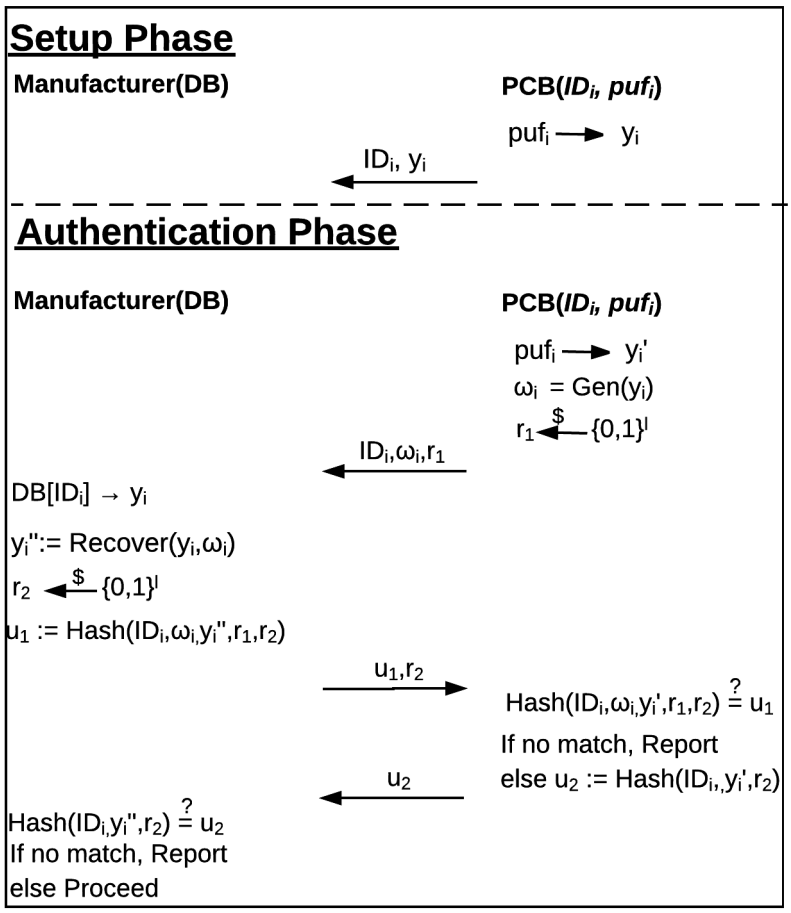


Figure 2.2: Mutual Authentication Protocol

2.2 Protocol Description

The mutual authentication protocol consists of two phases - the enrollment phase and the verification phase (as shown in Figure 2.2). During the enrollment phase, the manufacturer of the device enrolls one PUF response from the device into a secure database along with its serial number. This is a one-time operation and the interface used to collect the PUF response is destroyed immediately after enrollment.

Once enrollment is complete, the device is sent further down the supply chain to the user. At any time, a server with access to the secure database can initiate the mutual authentication

protocol. The server sends a message to the device requesting the serial ID and helper data. The device responds with the required information along with a random nonce (r_1). The helper data is used by the server to recover the noisy PUF response generated on the device. The server and device then authenticate each other using hashes computed from the recovered PUF response(u_1) and the noisy PUF response(u_2) respectively. These hash values are compared to determine device integrity.

2.3 Security Analysis

In this section, we present a security analysis of the mutual authentication protocol. We verify whether the protocol satisfies the security objectives outlined in Section 1.3.2.

- *Correctness* - The correctness property of the protocol depends on the number of error bits(n) in the noisy PUF response compared to the enrolled PUF response. The value of n should be less than that of the error correcting capability of the BCH code. If it is greater, then the Manufacturer will reject the device even if it is authentic(false reject). Also, the uniqueness of the PUF response should be greater than 50% to ensure that the Manufacturer does not accept faulty devices(false accept). We implement our protocol with the (255, 91, 25) BCH code. The protocol satisfies the correctness property as long as the number of errors in the noisy PUF response is 25 or less.

- *Confidentiality* - The PUF response remains a secret throughout the protocol. Any adversary listening on the communication channel will only receive helper data and hash values of the PUF response. Since the server controls the protocol runs, the number of helper data and hash values generated are also limited. The adversary will not be able to gain useful information by collecting these values as they reveal very little about the PUF response.

- *Resistance to Replay Attacks* - An adversary collecting protocol data cannot replay it

during a later run. This is due to the use of random nonces r_1, r_2 at the server and device side. These nonces ensure that the hash values are unique to each run of the protocol.

- *Resistance to Impersonation Attacks* - This protocol is resistant to impersonation attacks i.e., the adversary cannot successfully impersonate the server or the device during protocol operation. This is because the device and the server check for knowledge of the enrolled and noisy PUF responses respectively. Since an adversary will not be able to prove knowledge of either, the integrity check will fail.

In addition to the above properties, the protocol also ensures mutual authentication. The server proves its knowledge of the enrolled PUF response by sending the device a hash calculated from the reconstructed PUF response. The device uses this hash to authenticate the server before it reveals a hash of the noisy PUF response. This property is important because the helper data may be used to reveal information about the PUF response. By repeatedly requesting helper data, the adversary may try to guess the PUF response. Since the protocol supports mutual authentication, the device will be able to detect an invalid server and report this information to the user.

Chapter 3

Third-Party Verification Protocol

The Mutual Authentication protocol enables the manufacturer of a device to verify its integrity after the device is deployed. In many cases, the end user may wish to verify the integrity of his purchase. This cannot be achieved using the Mutual Authentication Protocol described in the previous chapter. It is because the PUF response enrolled in the secure database cannot be revealed to the end user. Otherwise, a malicious adversary may impersonate the end-user and gain access to sensitive PUF information. This problem (shown in Figure 3.1) is better expressed as follows: A third-party such as the end user must be able to match the noisy PUF response from the device with the PUF response enrolled in the Manufacturer's database. However, these values must not be revealed to the third party.

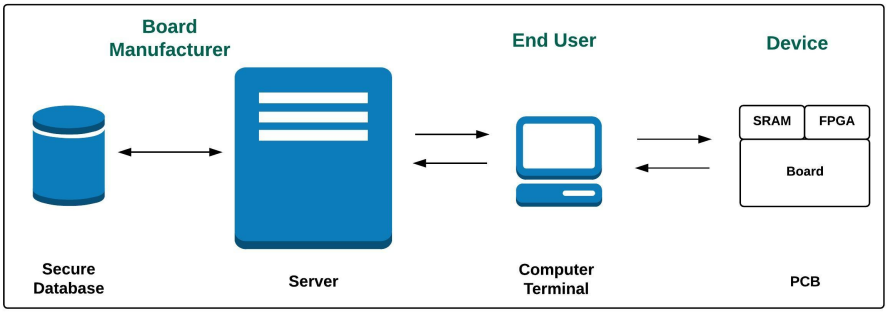


Figure 3.1: Topology for the Third-Party Verification Protocol

A straight forward approach to solving this problem is the use of keyed-hash message authentication codes(HMACs). The PUF response can be hashed down to a key of desired length. The third-party sends the same message to the device and the server. The device generates an HMAC over this message using the key extracted from the noisy PUF response. The server generates an HMAC over the same message using the key extracted from the PUF response enrolled in the secure database. The third-party then compares these two values to verify device integrity(Figure 3.2(a)).

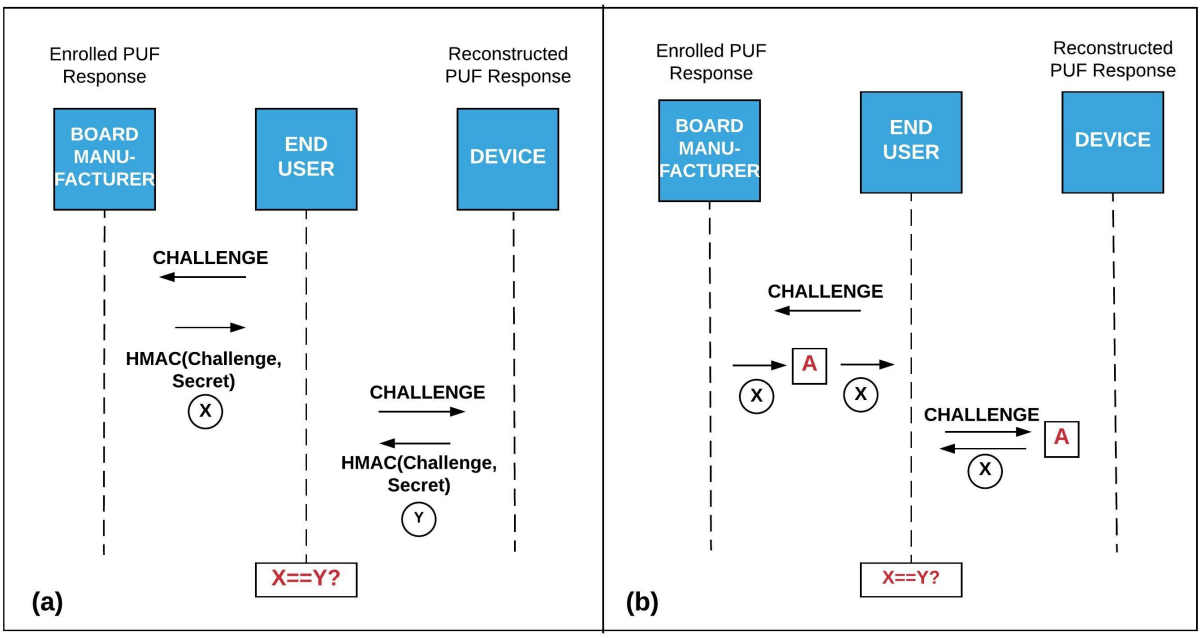


Figure 3.2: HMAC based approach and problem

However, this protocol does not guard against Man in the Middle(MiM) attacks. Consider the attack by an adversary as shown in the Figure 3.2(b). The adversary may collect the HMAC generated at the device and replay it as the server response. This will result in a false match at the third-party during the integrity check.

Our proposed protocol solves this problem of third-party verification of device integrity. We design our protocol using signature schemes to allow for light-weight implementations. In

this chapter, we give a detailed description of the proposed protocol and evaluate its security properties against the security objectives defined in Section 1.4.3. We also provide some preliminary knowledge about ECDSA signatures which is relevant to our implementation.

3.1 Notation

We use the following terms and notation while describing our third party verification protocol

- ID_i - Manufacturer serial number or board reference number in the database
- y_i - PUF response enrolled into the secure database
- puf_i - PUF embedded on the device
- ω_i - Helper Data
- y_i' - Noisy PUF Response
- d_i - Private Key for signatures
- SE - Strong Extraction
- $PK.Gen$ - Public Key generation from private key d_i
- Q_i - Public Key corresponding to the device private key d_i
- Rec - PUF Reconstruction using helper data and noisy PUF
- c - Random Challenge from the user
- r_1 - Random nonce
- h - Hash calculated by the device
- s - Signature over device hash h

- h' - Hash calculated by the user

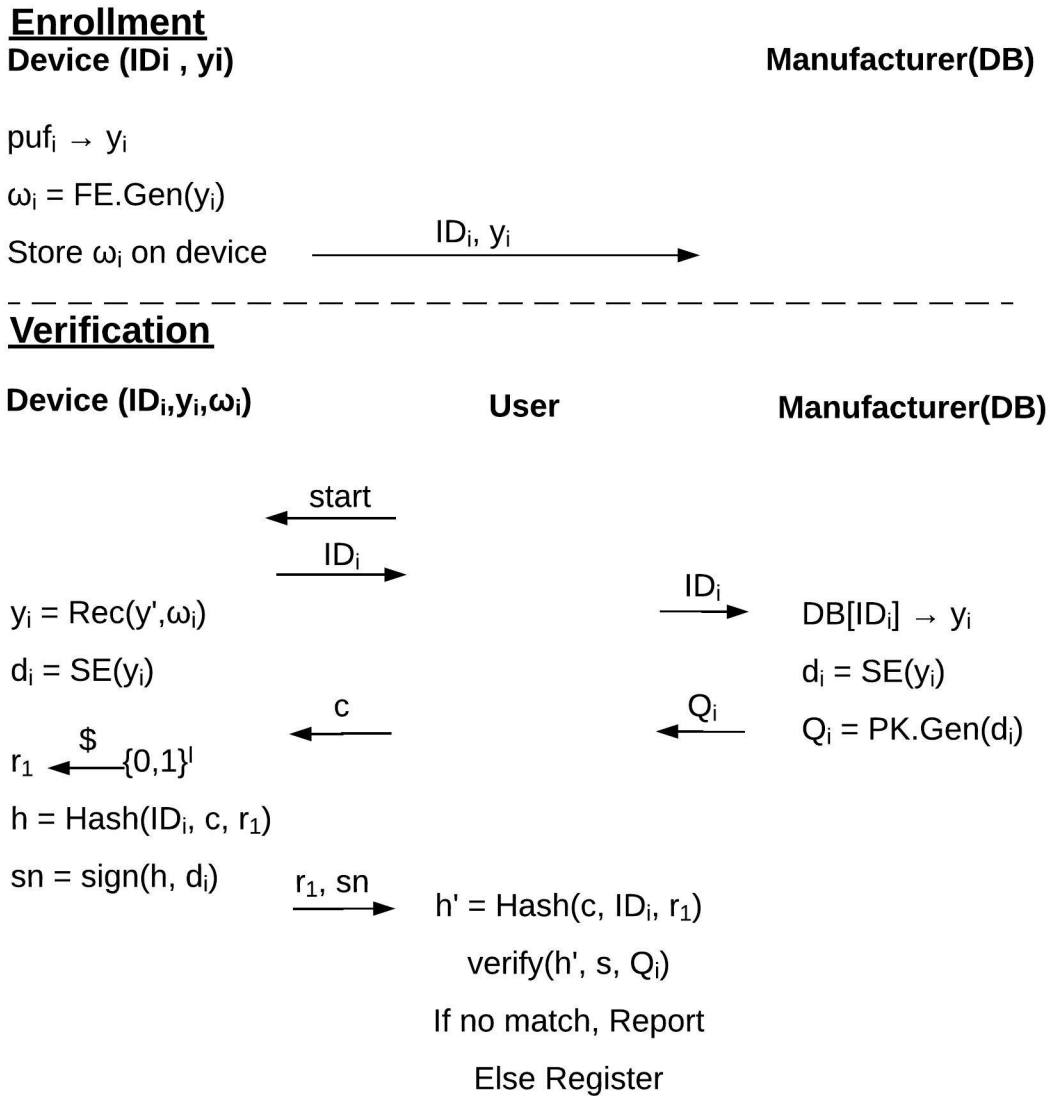


Figure 3.3: Third Party Verification Protocol

3.2 Protocol Description

The protocol is divided into two phases - Enrollment and Verification. Before deploying the device, the manufacturer enrolls its PUF response into a secure database. This operation is assumed to be secure. The device generates helper data corresponding to the enrolled PUF response and stores it in non-volatile memory on the device. This memory need not be secure as a single Helper Data value does not reveal any information about the PUF Response of the Device. In cases where there is no non-volatile memory available on the device, the helper data information can also be enrolled along with the PUF response. This information can be passed to the device when needed. After completing enrollment, the device is passed further down the supply chain to the user.

The user may initiate the verification phase at any time using a start message as shown in Figure 3.3. The Device responds with its serial identifier ID_i which is forwarded to the Manufacturer. The Device also starts generating a noisy PUF Response to reconstruct the PUF response generated during enrollment. This is achieved using fuzzy extraction and the Helper Data stored on the Device. The Private Key d_i is then generated from the reconstructed PUF response using strong extraction. Simultaneously, the Manufacturer retrieves the PUF response corresponding to the serial identifier (ID_i) given by the Device. A private key is extracted from the PUF response and a Public Key corresponding to the device is generated. This public key is sent to the user for the integrity check.

The User then sends a random challenge to the Device. The Device computes a hash over its ID, the received challenge and a random nonce. A signature scheme is used to sign the hash h to generate s. This signature s along with the random nonce r_1 is sent to the User. The User calculates the same hash h using the ID, c and r_1 values. The signature from the device is verified using this hash and the public key received from the Manufacturer. If the signature from the device corresponds to the public key from the manufacturer, the integrity check is successful. If the signature check fails, the device is detected as faulty and the User is notified about the same.

3.3 Security Analysis

In this section, we analyze the security properties of our protocol against the objectives that we defined in Section 1.4.3.

- *Correctness* - The proposed protocol satisfies the correctness property under certain conditions. For a genuine device to pass the integrity check, the signature generated by the device should be verified by the public key from the Manufacturer. If the noise in the PUF response of the device is greater than that of the error correcting capability of the BCH code, it will result in an incorrect key. This will result in a failed integrity check even if the device is authentic. The uniqueness of the Fusion PUF response should be greater than 50% so that false accepts do not occur and correctness is maintained.

- *Confidentiality* - The proposed protocol maintains confidentiality. PUF enrollment is a secure operation and the TCB on the device ensures that the adversary does not have access to the PUF response on the device. The public key, signature and random nonce values which are transmitted over the communication channel do not reveal any useful information about the PUF response. Any adversary controlling to the communication channel is unable to guess the PUF response by observing these values.

- *Resistance to Replay Attacks* - The protocol is resistant to replay attacks. The adversary cannot collect the signature from a previous protocol run and use it to authenticate a faulty device. This is because of the random nonce r_1 that is included in the hash. The signature generated by the device will be unique for each run of the protocol.

- *Resistance to Impersonation Attacks* - Unlike the mutual authentication protocol, the proposed protocol does not prevent User impersonation. However, since neither helper data nor PUF responses are revealed by the device or manufacturer, the adversary will not gain any value by impersonating the end user. Manufacturer impersonation is not possible without knowledge of the enrolled PUF response.

The proposed protocol can also be used in bootstrapping key infrastructure on networked devices. Bootstrapping is the procedure of enabling trust in a new device before key distribution is complete [18]. Existing solutions use pre-provisioning the keys on each device before deployment which is expensive and impractical. The third-party verification protocol provides a cost-effective and scalable solution to this bootstrapping problem. Upon booting up for the first time, the device tries to register with a domain in the environment. The domain can run the proposed protocol to validate that the device is genuine and then proceed with the registration.

3.4 ECDSA: Background

The Third Party Verification Protocol uses digital signatures as the underlying cryptographic operation. The efficiency and security of the proposed protocol depends on the signature scheme selected for use with this protocol. The key extracted from the noisy PUF is used as a private key for the signature . The public key can be generated according to the signature scheme that is used. El Gamal signatures use modular exponentiation to generate the signature as well as to compute the public key from the private key. Variants of this signature scheme include Schnorr signatures, Digital Signature Algorithm etc. Since the signature is generated on the device, we need the signature scheme to be efficient enough to run on a constrained platform. Therefore, we select ECDSA which is a DSA variant for elliptic curves as the underlying digital signature scheme for our implementation. ECDSA uses point multiplication which is a more efficient operation compared to modular multiplication [11]. In this section, we introduce concepts related to elliptic curve cryptography and provide some preliminary knowledge on generating and verifying digital signatures using ECDSA.

3.4.1 Elliptic Curve Cryptography

The security of cryptosystems like RSA is based on a trapdoor functions that are easy to compute in one direction but difficult to compute the inverse. RSA is a well known public-key operation that implements this trapdoor functionality by multiplying large primes. While multiplying two large primes is easy, factoring the product of two large primes is extremely difficult. However, as the availability of computational resources increases, the size of these prime numbers must also increase. This makes it difficult for highly constrained platforms like embedded systems to deploy RSA for their security applications. Elliptic curve cryptography(ECC) was introduced as a feasible alternative to RSA for such devices. Compared to RSA, the same level of security can be achieved with ECC using a smaller key. This translates to savings in power, time and computational resources which are critical to embedded systems. All these factors are influencing the adoption of ECC for security applications.

ECC is based on the unique properties of elliptic curves. It depends on the discrete logarithm problem that is hard to solve. Elliptic Curves are a set of points defined over a finite field that satisfy a mathematical equation given by:

$$y^2 = x^3 + ax + b \tag{3.1}$$

We define an operation called point multiplication over the points of an elliptic curve. Point multiplication is the repeated addition of a point on the curve such that all the intermediate results satisfy the curve equation. For example, consider two points $Q = (x_q, y_q)$ and $G = (x_g, y_g)$. Assume Q = p.G where "." refers to a point multiplication. This means that G is added n times to itself along the elliptic curve to arrive at Q. Even if we know the value of Q and G, we cannot detect the value of p. This is called the elliptic curve discrete logarithm problem and is the basis of ECC. The elliptic curve cryptosystem can be defined with three parameters - a curve equation, a generator point (G) and a maximum value for the points on the curve. The public key in such a cryptosystem would be Q and the private key would be p.

A finite field is a set of elements which satisfy some properties over addition and multiplication operations. If p is a prime number, we can define a finite field F_p such that 0, 1, 2 ... p-1 are all elements of the field. The addition and multiplication are modulo p operations. We define our elliptic curve over such a finite field where all point arithmetic is modulo p. The order(n) of the generator point is another important parameter as all operations in ECDSA are calculated using modulo n. Our implementation is based on elliptic curves defined over the Standards for Elliptic Curve Cryptography(SECG) and National Institute of Standards and Technology (NIST) primes.

Using these concepts of ECC, we can understand the operation of the Elliptic Curve Digital Signature Algorithm(ECDSA).

3.4.2 Introduction to ECDSA

ECDSA is an elliptic curve variation of the Digital Signature Algorithm(DSA). A digital signature is generated using the private key of the signer and is verified using the public key. It cannot be forged by an adversary who can read the signatures from a communication channel. In the following sections, we discuss the ECDSA Signature Generation, Verification algorithms.

ECDSA Signature Generation

To sign a message, we need the private key (d), the generator point (G), the order (n), a hash function (H), and a message (m). The result of the algorithm is the ECDSA signature (r,s). The steps involved in ECDSA signature generation are:

Step 1. Generate a uniformly random number k such that $k \epsilon$ [1, n-1]

Step 2: Compute a point $(x_1, y_1) = k.G$

Step 3: Compute $r = x_1 \ mod \ n$

Step 4: Calculate the hash e over the message $e = H(m)$ and collect the leftmost len bits where len is the number of bits used to represent the order of n.

Step 5: Compute $s = k^{-1}\ (e + dr)\ mod\ n$

The ECDSA signature pair is given by (r, s).

ECDSA Signature Verification

To verify a given ECDSA signature, we need the curve parameters(G,n), the signature pair (r,s), the input message (m) and the public key (Q). The steps involved in ECDSA signature verification are:

Step 1: Calculate the hash e over the message $e = H(m)$ and collect the leftmost len bits where len is the number of bits used to represent the order of n.

Step 2: Compute $w = s^{-1}\ mod\ n$

Step 3: Compute $u_1 = ew\ mod\ n$ and $u_2 = rw\ mod\ n$

Step 4: Compute the point $X(x_1, y_1) = u_1$ x G + u_2 x Q

Step 5: Verify signature validity by checking if the values of x_1, y_1 are less than maximum

Step 6: Verify whether $r = x_1\ mod\ n$

If the above statement is true, the given ECDSA signature pair (r, s) is detected as valid.

Chapter 4

Protocol Implementation

We evaluate the performance of the two protocols on two different hardware platforms - Altera DE2-115 and TI MSP430. The Altera FPGA implementation is based on a NIOS II soft-core whereas the TI MSP430 implementation uses the MSP430F5438A processor. This chapter follows in part [1] and provides a detailed description of our implementation.

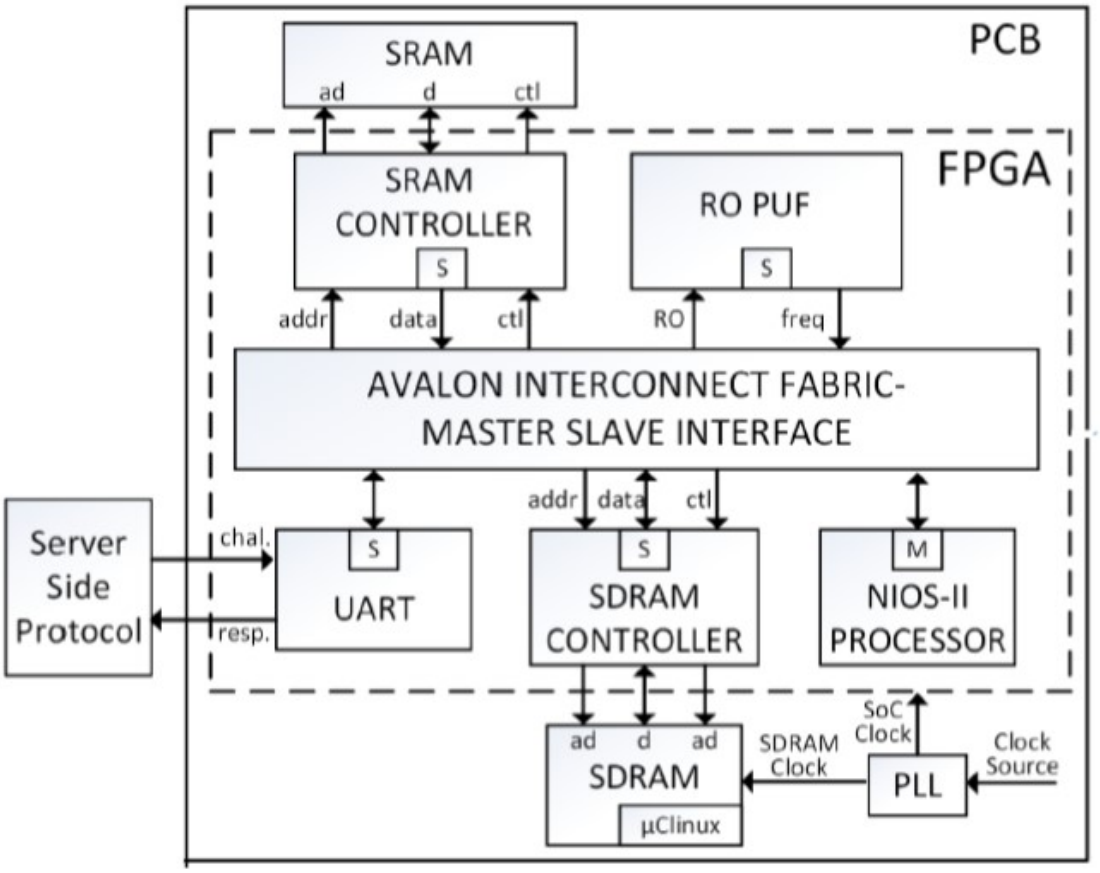
4.1 Altera DE2-115

This platform combines a large number of logic and I/O devices and can be used for a wide variety of applications. The logic devices on the board include an FPGA, SDRAM, SRAM and Flash. The I/O devices include an LCD display, LEDs, switches, RS-232 interface as well as connections to other devices.

4.1.1 Hardware Architecture

Figure 4.1 shows the Hardware Architecture of our prototype. Our design integrates the Nios-II softcore processor with SRAM, SDRAM, UART and several FPGA components(RO PUF, SRAM controller and SDRAM controller).

Figure 4.1: Altera DE2-115 Hardware Architecture



The hardware components in our design are connected using the Avalon Memory Mapped Interface which is an address based interface with master-slave connections. The Nios-II processor controls the signals of the data slave components (SRAM Controller, RO-PUF, SDRAM Controller, JTAG) and of instruction slave components (SDRAM Controller). The PUF data is collected by the software using Memory Mapped Registers. The hardware collects the input arguments to the PUF from these registers and the output is placed back into the memory registers corresponding to the PUF. We demonstrate the Fusion PUF as a combination of the SRAM PUF and Ring-Oscillator PUF(RO-PUF).

SRAM PUF Design: The SRAM PUF is based on the preferred power-up state of transistor cells in an SRAM. Our implementation uses the SRAM on the Altera DE2-115 board (ISSI

IS61WV102416BLL) and an FSM controller. The SRAM controller is interfaced to the SoC as a slave, and it handles communication between the Nios-II processor and the 1024k x 16 bit CMOS SRAM chip. The controller receives the address of a memory location in SRAM, reads the startup value at that address location and transmits the values to the processor through the Avalon Fabric.

RO-PUF Design: The RO-PUF exploits delay variations in the logic elements to produce a unique n-bit identifier [10]. The design works by chaining an odd number of inverters and connecting the output of the last inverter to the input of the first inverter. This way, each RO produces an oscillating output with a frequency dependent on how quickly the looping signal propagates through the logic elements. The output (frequency) of an RO (A) is compared to the output of another RO (B), and either a 1 or a 0 is produced depending on whether A or B has a faster frequency. In our design, we use 256 ROs to generate a 255 bit PUF output. To read stabilized RO frequencies, the system waits for 100 ms before starting the measurement. Then, it starts counting the oscillations for 250 ms.

We use the same Hardware Architecture to demonstrate the operation of both the protocols.

4.1.2 Software Architecture

This section explains the Software Architecture used in our prototype. While the same software design is retained, there is a difference in the functionality of software modules between the two protocols. We also provide a brief introduction to the OpenSSL library functions relevant to the implementation of the ECDSA signature scheme.

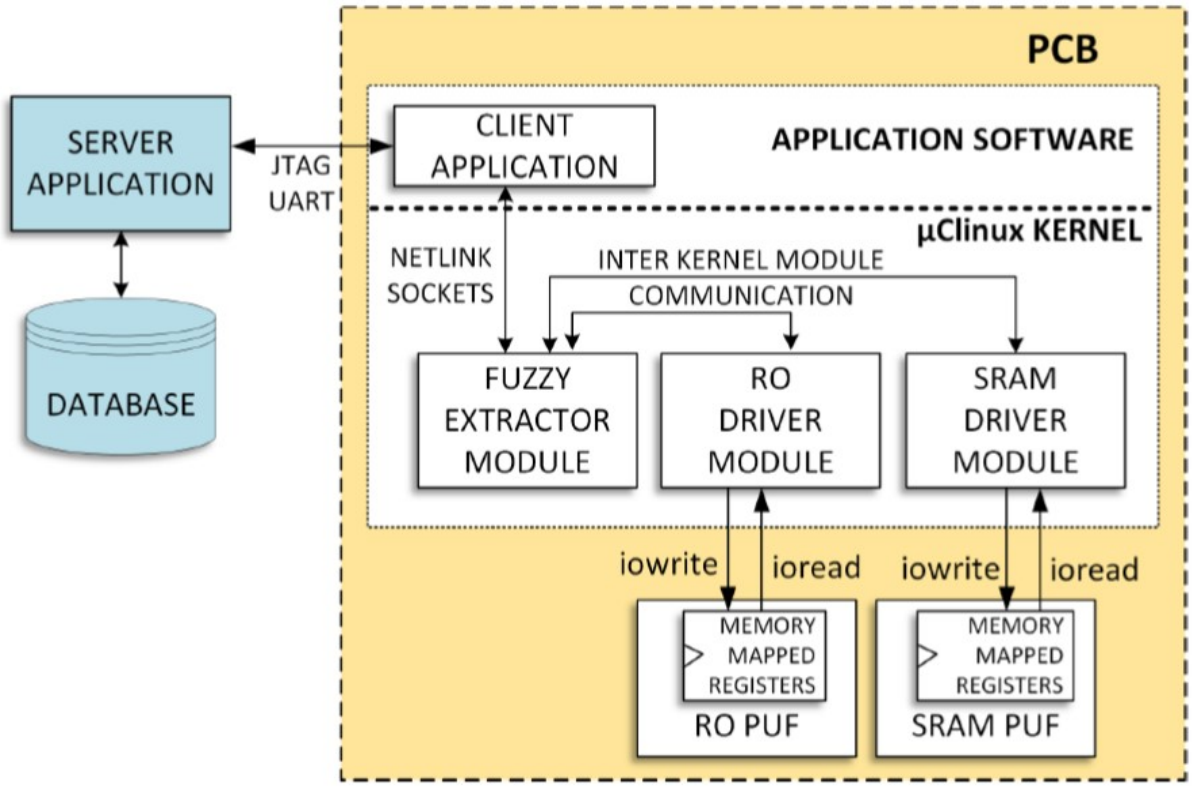


Figure 4.2: Altera DE2-115 Software Architecture for the Mutual Authentication Protocol

Mutual Authentication Protocol

Figure 4.2 shows the software architecture of our prototype. It consists of three components: (i) the μClinux operating system with kernel modules, (ii) the Client Application on the device that interacts with the Manufacturer Server, and (iii) the Host Application used by the Server to authenticate the PCB. These components implement the Mutual Authentication Protocol defined in Chapter 2. The use of μClinux on a Nios-II was done to demonstrate the integration of the Fusion PUF as a combination of hardware and software, and to support easy integration with a networking stack. We consider integration of a TCB as a future effort. The protocol needs a secure method to collect the PUF data from the PCB components and send it to the verifier. Furthermore, the PUF data is privileged information throughout

its processing. In our prototype, PUF-related computing is entirely carried out within the μClinux kernel and no user level application has access to this data.

Table 4.1: Data flow during the Mutual Authentication protocol execution

Protocol Operation	Server App	Client App	FE Mod	RO PUF Driver	SRAM Driver
	Req				
$\text{puf}_i \rightarrow y'_i$				Read	Read
$\omega_i := \text{Gen}(y'_i)$			Comp		
$r_1 \xleftarrow{\$} \{0,1\}^l$			Comp		
ID_i, ω_i, r_1		Send			
ID_i, ω_i, r_1	Recv				
$\text{DB}[ID_i] \rightarrow y_i$	Comp				
$y''_i := \text{Recover}(y_i, \omega_i)$	Comp				
$r_2 \xleftarrow{\$} \{0,1\}^l$	Comp				
$u_1 := \text{Hash}(...)$	Comp				
u_1, r_2	Send				
u_1, r_2		Recv			
Hash(...)			Comp		
$\stackrel{?}{=} u_1$		**Auth**			
$u_2 := \text{Hash}(...)$			Comp		
$u2$		Send			
$u2$	Recv				
Hash(...)	Comp				
$\stackrel{?}{=} u_2$	**Auth**				

We designed three kernel modules to implement the PUF post-processing and the sensitive steps of the authentication protocol. These modules are the Fuzzy Extractor module, the SRAM driver, and the RO driver. Table 4.1 summarizes the operations of these modules.

The SRAM PUF driver maps the SRAM into the μClinux kernel memory space, so that it can be directly read from the Fuzzy Extractor Module. The RO PUF driver generates a PUF signature by comparing the value of 256 ring oscillators on the FPGA. The frequency values for the ROs are read from a memory mapped location. These values are compared to generate the PUF response for the RO PUF.

The Fuzzy Extractor Module interacts with the SRAM and RO PUF drivers to generate the PUF response as shown in Table I. This response is then processed within the Fuzzy Extractor to generate the helper data that is sent to the Server. The module is also responsible for generating the SHA-256 hashes (u_1, u_2 Table 4.1) used in the protocol. The Client application runs on the μClinux operating system on the device. This entity is responsible for receiving and processing messages from the Server. It receives the messages sent by the Host Application over JTAG UART, creates appropriate requests to the kernel based on the received message and sends it back to the Host Application.

Third Party Verification Protocol

The overall software architecture for the Third-party verification protocol is similar to that of the Mutual Authentication Protocol. However, the device interacts with the User application which also interfaces with an application simulating the Manufacturer's Server. The functionality of individual modules in the diagram are also different. This can be seen in Table 4.2.

Table 4.2: Data flow during the Third Party Verification Protocol execution

Protocol Operation	User App	Client App	FE Mod	RO PUF Driver	SRAM Driver	Manufacturer
	Req					
ID_i		Send				
ID_i	Recv/Fwd					Recv
$\text{puf}_i \rightarrow y'_i$				Read	Read	
$y_i := \text{Recover}(y'_i, \omega_i)$			Comp			
$d_i = \text{SE}(y_i)$			Comp			
$\text{DB}[ID_i] \rightarrow y_i$						Comp
$d_i = \text{SE}(y_i)$						Comp
$Q_i = \text{PK.Gen}(d_i)$	Recv					Comp/Send
c	Send	Recv				
$r_1 \xleftarrow{\$} \{0,1\}^l$			Comp			
$h = \text{Hash}(ID_i, c, r_1)$			Comp			
$sn = \text{ecdsa\_sign}(h, d_i)$		Comp				
sn, r_1	Recv	Send				
$h' = \text{Hash}(c, ID_i, r_1)$	Comp					
$\text{ecdsa\_verify}(h', sn, Q_i)$	**Auth**					

Similar to the mutual authentication protocol, we have three modules in the kernel - the RO PUF module, the SRAM PUF module and the Reverse Fuzzy Extractor Module. The Fuzzy Extractor module interacts with the Client Application on the device to carry out protocol operations. It also interacts with the SRAM PUF and RO PUF modules to generate PUF responses. The SRAM PUF and RO PUF modules implement the same functionality as the previous protocol.

The Fuzzy Extractor Module generates a noisy PUF response and reconstructs the enrolled

PUF response using the helper data information stored on the device. It then extracts a private key from the reconstructed PUF response using strong extraction based on the SHA-256 hash function. The processing of the PUF response and the strong extraction is confined to the Fuzzy Extractor module within the OS kernel. These operations are secured by the OS running on a TCB. The Client Application uses the OpenSSL cryptographic library to generate the ECDSA signature. The User Application interacts with the Server Application to retrieve the public key (Q) associated with the device and verifies the signature generated by the device using this public key.

The ECDSA signature scheme forms a major part of the Third-Party Verification Protocol. For the Altera DE2-115 Implementation, we use the OpenSSL cryptographic tool-kit to perform ECDSA operations. It is an open source library which supports a number of cryptographic algorithms.

OpenSSL

The OpenSSL EC library provides API for Elliptic Curve Operations. This is used to implement ECDSA and other algorithms. For 80-bit security, we need the key size for ECC to be at least 160 bits. We use the NIST-P256 curve for our implementation which is a NIST recommended curve over a 256 bit prime field. The OpenSSL code for the ECDSA algorithm is given in the Appendix.

4.2 TI MSP430

Micro-controllers are used in a wide variety of applications ranging from Wireless Sensor Nodes to Industrial Control Devices. Such devices are deployed in large numbers to locations that may not be easily accessible. It is desirable to enable remote integrity checks in such scenarios so that any tampering of devices can be detected. The power, computational and memory constraints on micro-controller based platforms create the need for an efficient and

low-cost integrity check.

We implemented the two protocols on the TI MSP430 family of microcontrollers to evaluate the feasibility of their deployment on such devices. The following sections give a detailed description of our prototype. We discuss the hardware capabilities of our platform followed by an explanation of the software architecture.

4.2.1 Hardware Architecture

The MSP430 family of microcontrollers from Texas Instruments are designed for ultra low power applications. The MSP430F5438A Experimenter Board used in our prototype features a powerful 16-bit RISC CPU, 256 KB Flash, 16 KB RAM, 4 USCIs and a 32-bit Hardware Multiplier. The device is connected to a User Application running on a computer through a USB-Serial Interface.

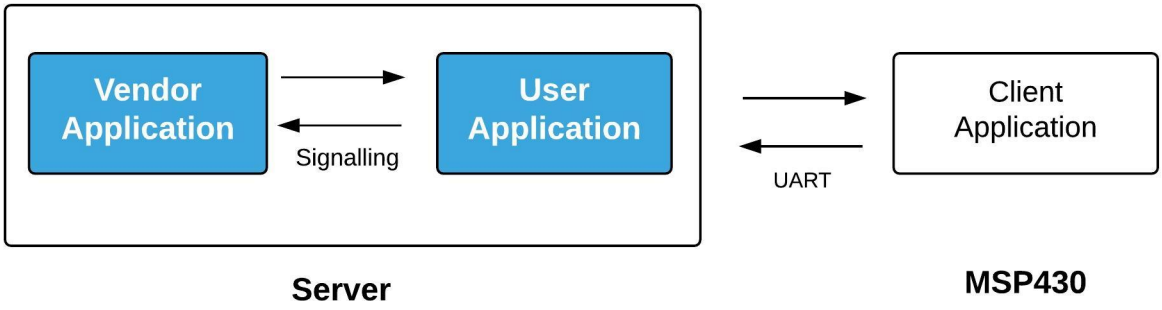


Figure 4.3: Software Architecture for the Mutual Authentication Protocol running on MSP430

4.2.2 Software Architecture

Figure 4.3 gives an overview of the software modules in our design. The architecture is same for both protocol implementations. However, the functionality of individual modules

differs. We have a Client Application running on the device which interfaces with a Computer terminal simulating the Manufacturer Server/ End User Server Application. In the following sections, we provide details of our implementation for the two protocols.

Mutual Authentication Protocol

We follow the Mutual Authentication Protocol as described in Chapter 2. The Client Application running on the device can be further divided into two modules - the main application carrying out protocol operations and the BCH library that generates code words for fuzzy extraction. The Client Application interacts with the Server Application and generates helper data and Hashes(u_1 and u_2) for server and device authentication. We do not include PUF response generation as part of our implementation. Modern micro-controllers include an SRAM module on the device that can be used to generate a PUF response without adding much overhead to our current implementation [2]. The Server Application interfaces with the device by reading/writing to a COM port exposed by the device. This Application can be run on a Windows/Linux OS provided the USB-Serial Driver for the MSP430 device is installed on the OS.

Third-Party Verification Protocol

The Software Architecture consists of three modules - the Client Application running on the device, the applications simulating End User and Manufacturer Server behavior running on a Windows/Linux based Computer. The End User Application interfaces with the device by reading/writing to the COM port. The Client Application uses the helper data stored on the device to reconstruct the PUF response generated during enrollment. This reconstructed response is used to generate the private key for ECDSA using strong extraction. The ECDSA signature is generated using this private key and the RELIC crypto-toolkit. The End User Application then interacts with the Server Application to retrieve the public key (Q) used to verify the signature generated by the device.

The ECDSA signature scheme forms a major portion of the Third-Party Verification protocol implementation. Since the TI MSP430 is a highly constrained platform, we use the RELIC cryptographic tool-kit for the ECDSA operations. RELIC is designed specifically for efficient and flexible implementations of cryptographic algorithms.

RELIC

RELIC is an efficient library for cryptographic operations. It is easily portable to various platforms and includes architecture dependent code for efficient implementations. The RELIC tool-kit is licensed under LGPL v2.1 and supports a wide variety of crypto algorithms including multi-precision integer arithmetic, elliptic curves over prime and binary fields, prime and binary field arithmetic etc as well as various cryptographic protocols. Our configuration of RELIC is tailored to implement the ECDSA algorithm. The RELIC source code for the implementation of the ECDSA signature scheme is provided in the Appendix.

Chapter 5

Results

In this chapter, we present our results from running the Mutual Authentication and the Third Party Verification protocols on the Altera DE2-115 and TI MSP430 platforms. We evaluate the performance and memory requirements for the protocol implementations to determine the feasibility of their use.

5.1 Altera DE2-115

This section details our hardware and software implementation results for both the protocols on DE2-115.

5.1.1 Hardware Implementation Costs

Table 5.1 shows the hardware utilization for the design implementation.

Results obtained demonstrate that Construction of 256 bit RO PUF consumes 50% of the total resources for the system design. The design makes use of Altera Quartus II software version 15.0. Additionally, the design utilizes the Logic Lock constraint generation tool for

Table 5.1: Hardware Utilization for FPGA Components on DE2-115

Module	LE	BRAM	DSP
Nios-II	2254	8	4
RO-PUF	4385	-	-
SRAM controller	5	-	-
SDRAM controller	327	-	-
JTAG-UART	146	1	-
PLL	8	-	-
Total	7125	9	4

the RO PUF. The entire FPGA system is designed using Qsys system integration tool. The same hardware design is used for the demonstration of the Third Party Verification Protocol.

5.1.2 Software Implementation Costs

Mutual Authentication Protocol

Table 5.2 lists the Software implementation cost of the design for the Mutual Authentication Protocol. The host is a x64-based Windows 10 PC with Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz. The host application is compiled using GNU Compiler Collection (GCC) version 4.9.3 in a Cygwin environment and uses the jtag_atlantic.dll import library to access the JTAG UART.

Table 5.2: DE2-115 Memory Requirements of Software Components of the Mutual Authentication Protocol

Module	Codesize (Bytes)	Ratio (%)
RO-PUF	38,820	2.28
SRAM PUF	35,128	2.07
Fuzzy Extractor	116,680	6.86
Board Application	35,360	2.08
μClinux Kernel	1,475,069	100

The total time taken for the execution of the Mutual Authentication protocol on the DE2-115 is 111.72s. Table 5.3 shows the breakdown of the PCB side response time at each protocol operation. It is evident from the table that most of the overall protocol response time is spent in RO-PUF response generation.

Table 5.3: Breakdown of the Mutual Authentication Protocol Operation

Authentication Step	Response Time	Unit
SRAM Read	249	μs
RO Read	89.2	s
Overall PUF Generation	89.23	s
BCH Encoding	25.02	ms
Random Number Generator (91 bits)	12.17	ms
u_1 Hash	2.15	ms
u_2 Hash	0.28	ms
Overall PUF processing	89.32	s

Third Party Verification Protocol

The memory and computational resources utilized for generating the ECDSA signature depend mainly on the Elliptic Curve selected for the ECDSA algorithm. The OpenSSL crypto tool-kit is used to carry out the ECDSA signature generation for the Third Party Verification Protocol. We use the NIST-P256 curve for the DE2-115 implementation as it has higher memory and computational capabilities when compared to MSP430. The parameters of the standard NIST-P256 curve are given by:

a = -3

b = 5ac635d8 aa3a93e7 b3ebbd55 769886bc 651d06b0 cc53b0f6 3bce3c3e 27d2604b

p = 115792089210356248762697446949407573530086143415290314195533631308867097853951

The x and y coordinates of the base point are:

G_x = 6b17d1f2 e12c4247 f8bce6e5 77037d81 2deb33a0 f4a13945 d898c296

G_y = 4fe342e2 fe1a7f9b 8ee7eb4a 2bce3357 6b315ece cbb64068 37bf51f5

Table 5.4 lists the memory sizes of the Software Modules used in our implementation. There is an increase in the size of the Fuzzy Extractor Module due to the BCH decoder required for the Reconstruction of the enrolled PUF response. As the Board Application also incorporates ECDSA functionality from the OpenSSL library, we see a significant increase in its size.

Table 5.4: DE2-115 Memory Requirements of Software Components of the Third Party Verification Protocol

Module	Codesize (Bytes)	Ratio (%)
RO-PUF	38,820	2.33
SRAM PUF	35,128	2.11
Fuzzy Extractor	163,455	9.81
Board Application	446,664	26.8
μClinux Kernel	1,664,775	100

Table 5.5 shows the amount of time spent on each Protocol Operation. A major part of the protocol time is spent on generating the RO PUF response. The ECDSA signature generation is the second most time consuming operation.

Table 5.5: Breakdown of the Third Party Verification Protocol Operation

Authentication Step	Response Time	Unit
SRAM Read	249	μs
RO Read	89.2	s
Overall PUF Generation	89.23	s
BCH Decoding	33.76	ms
Strong Extraction	127	μs
ECDSA Signature	237	ms

5.2 TI MSP430

The following sections present the results of our implementation on the MSP430F5438A Experimenter board. We measure the cycle counts of the protocol operations and memory costs of software modules used in the implementation.

5.2.1 Mutual Authentication Protocol

Table 5.6 shows the memory costs of the various software modules running on the device. The RELIC Crypto-Toolkit is configured to generate the random numbers and cryptographic hashes for the protocol.

Table 5.6: Memory Requirements for MSP430 Software running the Mutual Authentication Protocol

Software Component	Size (in bytes)
RELIC Crypto-Toolkit	14,152
BCH Encoder	9,378
UART	1,209
Overall Size of the Application	34,802

Table 5.7 lists the cycle count for each protocol operation on MSP430F5438A. The random number generator operation of RELIC consumes the maximum amount of CPU cycles. RELIC is configured to use the FIPS 186-2 SHA-1 based generator. Each random bit generated requires one SHA-1 computation. This is reflected in the results seen in the table.

Table 5.7: MSP430 Cycle Counts for Mutual Authentication Protocol Operations

Verification Step	Number of Cycles
Generate r_1 (32 bits)	7,806,618
Generate random number for helper data(91 bits)	73,610,673
BCH Encode	224,650
Compute Hash u_2	225,626

5.2.2 Third Party Verification Protocol

The number of clock cycles depends greatly on the curve selected for the ECDSA signature scheme. For an 80 bit security level, it is recommended to use a key size of atleast 160 bits. We use the SECG_P160 curve in our implementation. The curve parameters are as follows:

The curve E: $y^2 = x^3 + ax + b$ over F_p is defined by:

p = = $2^{160} - 2^{31} - 1$

a = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF 7FFFFFFC

b = 1C97BEFC 54BD7A8B 65ACF89F 81D4D4AD C565FA45

The base point G in compressed form is:

G = 02 4A96B568 8EF57328 46646989 68C38BB9 13CBFC82

Table 5.8 lists the memory costs of implementing various software modules on MSP430F5438A. The increase in the size of the RELIC library is due to the addition of Elliptic Curve Cryptography and ECDSA functionality. The BCH Decoder consumes more memory resources compared to the Encoder. This is due to the memory required to compute the error location polynomial and the increased complexity of the code. This results in an overall increase (around 11KB) in terms of code size when compared to the Mutual Authentication Protocol.

Table 5.8: Memory Requirements for MSP430 Software running the Third Party Verification Protocol

Software Component	Size (in bytes)
RELIC Crypto-Toolkit	22,454
BCH Decoder	12,126
UART	1,209
Overall Size of the Application	46,191

Table 5.9: MSP430 Cycle Counts for the Third Party Verification Protocol Operations

Verification Step	Number of Cycles
BCH Decoder	4,424,239
Strong Extraction	84,211
ECDSA Signature Generation	62,016,964

Table 5.9 lists the cycle counts for each step of the Third Party Verification Protocol. The ECDSA signature generation has the highest cycle count. It involves one point multiplication, three modular reductions, one SHA-1 computation, one scalar multiplication, one scalar addition and two inverse computations. One point multiplication operation in RELIC takes 61,095,024 cycles. This operation can be performed in less than 6 million cycles(MSP430) by using assembly optimized versions of point multiplication [20].

Chapter 6

Conclusion

In this thesis, we designed and implemented PUF based integrity verification protocols on two platforms. First, we identified various security objectives relevant to our application scenario and analyzed the security properties of the Mutual Authentication protocol. Then, we proposed a Third Party Verification protocol that enables End Users to perform integrity checks without knowledge of the PUF responses. We evaluated the security properties of the proposed protocol against our security objectives.

In the second part, we implement the protocols on two different platforms - Altera DE2-115 and TI MSP430. The demonstration using μClinux and the low software and hardware overhead on DE2-115 prove that PUF based verification techniques can be easily integrated into scalable solutions. Our implementation on the TI MSP430 platform incurs a software overhead of 50 kB. While the cycle count for the ECDSA algorithm is high, we see that this can be easily optimized using architecture specific code for point multiplications.

Bibliography

[1] A. Aysu et al. "A design method for remote integrity checking of complex PCBs". In: *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2016, pp. 1517–1522.

[2] C. Bohm, M. Hofer, and W. Pribyl. "A microcontroller SRAM-PUF". In: *Network and System Security (NSS), 2011 5th International Conference on*. Sept. 2011, pp. 269–273. DOI: 10.1109/ICNSS.2011.6060013.

[3] J. Lawrence Carter and Mark N. Wegman. "Universal Classes of Hash Functions (Extended Abstract)". In: *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*. STOC '77. Boulder, Colorado, USA: ACM, 1977, pp. 106–112. DOI: 10.1145/800105.803400. URL: http://doi.acm.org/10.1145/800105.803400.

[4] W. Che, F. Saqib, and J. Plusquellic. "PUF-based authentication". In: *Computer-Aided Design (ICCAD), 2015 IEEE/ACM International Conference on*. Nov. 2015, pp. 337–344. DOI: 10.1109/ICCAD.2015.7372589.

[5] P. F. Cortese et al. "Efficient and practical authentication of PUF-based RFID tags in supply chains". In: *RFID-Technology and Applications (RFID-TA), 2010 IEEE International Conference on*. June 2010, pp. 182–188. DOI: 10.1109/RFID-TA.2010.5529941.

[6] Yevgeniy Dodis et al. "Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data". In: *SIAM J. Comput.* 38.1 (Mar. 2008), pp. 97–139. ISSN: 0097-5397. DOI: 10.1137/060651380. URL: http://dx.doi.org/10.1137/060651380.

[7] SIA Anti-Counterfeiting Task Force. *Winning the battle against counterfeit semiconductor products*. Tech. rep. Semiconductor Industry Association, Aug. 2013.

[8] Y. Gao et al. "Obfuscated challenge-response: A secure lightweight authentication mechanism for PUF-based pervasive devices". In: *2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*. Mar. 2016, pp. 1–6. DOI: 10.1109/PERCOMW.2016.7457162.

[9] Blaise Gassend et al. "Silicon Physical Random Functions". In: *Proceedings of the 9th ACM Conference on Computer and Communications Security*. CCS '02. Washington, DC, USA: ACM, 2002, pp. 148–160. ISBN: 1-58113-612-9. DOI: 10.1145/586110.586132. URL: http://doi.acm.org/10.1145/586110.586132.

[10] Jorge Guajardo et al. "FPGA Intrinsic PUFs and Their Use for IP Protection". In: *Cryptographic Hardware and Embedded Systems - CHES 2007: 9th International Workshop, Vienna, Austria, September 10-13, 2007. Proceedings*. Ed. by Pascal Paillier and Ingrid Verbauwhede. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 63–80. ISBN: 978-3-540-74735-2. DOI: 10.1007/978-3-540-74735-2_5. URL: http://dx.doi.org/10.1007/978-3-540-74735-2_5.

[11] Nils Gura et al. "Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs". In: *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*. Ed. by Marc Joye and Jean-Jacques Quisquater. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 119–132. ISBN: 978-3-540-28632-5. DOI: 10.1007/978-3-540-28632-5_9. URL: http://dx.doi.org/10.1007/978-3-540-28632-5_9.

[12] Ghaith Hammouri, Erdinç Öztürk, and Berk Sunar. "A Tamper-proof and Lightweight Authentication Scheme". In: *Pervasive Mob. Comput.* 4.6 (Dec. 2008), pp. 807–818.

ISSN: 1574-1192. DOI: 10.1016/j.pmcj.2008.07.001. URL: http://dx.doi.org/10.1016/j.pmcj.2008.07.001.

[13] S. W. Jung and S. Jung. "HRP: A HMAC-based RFID mutual authentication protocol using PUF". In: *The International Conference on Information Networking 2013 (ICOIN)*. Jan. 2013, pp. 578–582. DOI: 10.1109/ICOIN.2013.6496690.

[14] F. Koushanfar and G. Qu. "Hardware metering". In: *Design Automation Conference, 2001. Proceedings*. 2001, pp. 490–493. DOI: 10.1109/DAC.2001.156189.

[15] Roel Maes. "PUF-Based Entity Identification and Authentication". In: *Physically Unclonable Functions: Constructions, Properties and Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 117–141. ISBN: 978-3-642-41395-7. DOI: 10.1007/978-3-642-41395-7_5. URL: http://dx.doi.org/10.1007/978-3-642-41395-7_5.

[16] Mehrdad Majzoobi et al. "Slender PUF Protocol: A Lightweight, Robust, and Secure Authentication by Substring Matching". In: *Proceedings of the 2012 IEEE Symposium on Security and Privacy Workshops*. SPW '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 33–44. ISBN: 978-0-7695-4740-4. DOI: 10.1109/SPW.2012.30. URL: http://dx.doi.org/10.1109/SPW.2012.30.

[17] Noam Nisan and David Zuckerman. "Randomness is Linear in Space". In: *J. Comput. Syst. Sci.* 52.1 (Feb. 1996), pp. 43–52. ISSN: 0022-0000. DOI: 10.1006/jcss.1996.0004. URL: http://dx.doi.org/10.1006/jcss.1996.0004.

[18] Max Pritikin et al. *Bootstrapping Key Infrastructures*. Internet-Draft draft-ietf-anima-bootstrapping-keyinfra-02. Work in Progress. Internet Engineering Task Force, Mar. 17, 2016. 38 pp. URL: https://tools.ietf.org/html/draft-ietf-anima-bootstrapping-keyinfra-02.

[19] IHS Technology. *Top 5 Most Counterfeited Parts Represent a $169 Billion Potential Challenge for Global Semiconductor Market @ONLINE*. Apr. 2012. URL: https://technology.ihs.com/405654/top-5-most-counterfeited-parts-represent-a-169-billion-potential-challenge-for-global-semiconductor-market.

[20] Erich Wenger, Thomas Unterluggauer, and Mario Werner. "8/16/32 Shades of Elliptic Curve Cryptography on Embedded Processors". In: *Progress in Cryptology – INDOCRYPT 2013: 14th International Conference on Cryptology in India, Mumbai, India, December 7-10, 2013. Proceedings*. Ed. by Goutam Paul and Serge Vaudenay. Cham: Springer International Publishing, 2013, pp. 244–261. ISBN: 978-3-319-03515-4. DOI: 10.1007/978-3-319-03515-4_16. URL: http://dx.doi.org/10.1007/978-3-319-03515-4_16.

Appendix A

Program Source

This chapter shows code listings for the ECDSA signature scheme using RELIC and OpenSSL.

Listing A.1: ECDSA Signature Generation using RELIC

```
void cp_ecdsa_sig(bn_t r, bn_t s, unsigned char *msg, int len, int hash,bn_t
        bn_t n, k, x, e;
        ec_t p;
        unsigned char h[MD_LEN];

        bn_null(n);
        bn_null(k);
        bn_null(x);
        bn_null(e);
        ec_null(p);

        TRY {
            bn_new(n);
            bn_new(k);
            bn_new(x);
            bn_new(e);
            ec_new(p);

            ec_curve_get_ord(n);
            do {
                do {
                    do {
                        bn_rand(k, BN_POS, bn_bits(n));
```

```
                    bn_mod(k, k, n);
                } while (bn_is_zero(k));

                ec_mul_gen(p, k);
                ec_get_x(x, p);
                bn_mod(r, x, n);
            } while (bn_is_zero(r));

            if (!hash) {
               md_map(h, msg, len);
               msg = h;
               len = MD_LEN;
            }
            if (8 * len > bn_bits(n)) {
                len = CEIL(bn_bits(n), 8);
                bn_read_bin(e, msg, len);
                bn_rsh(e, e, 8 * len - bn_bits(n));
            } else {
                bn_read_bin(e, msg, len);
            }

            bn_mul(s, d, r);
            bn_mod(s, s, n);
            bn_add(s, s, e);
            bn_mod(s, s, n);
            bn_gcd_ext(x, k, NULL, k, n);
            if (bn_sign(k) == BN_NEG) {
                bn_add(k, k, n);
            }
            bn_mul(s, s, k);
            bn_mod(s, s, n);
        } while (bn_is_zero(s));
    }
    CATCH_ANY {
       THROW(ERR_CAUGHT);
    }
    FINALLY {
        bn_free(n);
        bn_free(k);
        bn_free(x);
            bn_free(e);
                    ec_free(p);
      }
}
```

Listing A.2: ECDSA Signature Verification using RELIC

```
int cp_ecdsa_ver(bn_t r, bn_t s, unsigned char *msg, int len, int hash, ec_t
        bn_t n, k, e, v;
        ec_t p;
        unsigned char h[MD_LEN];
        int result = 0;

        bn_null(n);
        bn_null(k);
        bn_null(e);
        bn_null(v);
        ec_null(p);

      TRY {
          bn_new(n);
          bn_new(e);
          bn_new(v);
          bn_new(k);
          ec_new(p);

          ec_curve_get_ord(n);

          if (bn_sign(r) == BN_POS && bn_sign(s) == BN_POS &&
                  !bn_is_zero(r) && !bn_is_zero(s)) {
              if (bn_cmp(r, n) == CMP_LT && bn_cmp(s, n) == CMP_LT) {
                  bn_gcd_ext(e, k, NULL, s, n);
                  if (bn_sign(k) == BN_NEG) {
                      bn_add(k, k, n);
                  }

                  if (!hash) {
                     md_map(h, msg, len);
                     msg = h;
                     len = MD_LEN;
                  }
                  if (8 * len > bn_bits(n)) {
                      len = CEIL(bn_bits(n), 8);
                      bn_read_bin(e, msg, len);
                      bn_rsh(e, e, 8 * len - bn_bits(n));
                  } else {
                      bn_read_bin(e, msg, len);
                  }

                  bn_mul(e, e, k);
```

```
                bn_mod(e, e, n);
                bn_mul(v, r, k);
                bn_mod(v, v, n);

                ec_mul_sim_gen(p, e, q, v);
                ec_get_x(v, p);

                bn_mod(v, v, n);

                result = dv_cmp_const(v->dp, r->dp, MIN(v->used, r->use
                result = (result == CMP_NE ? 0 : 1);

                if (v->used != r->used) {
                    result = 0;
                }

                if (ec_is_infty(p)) {
                    result = 0;
                }
            }
        }
    }
    CATCH_ANY {
        THROW(ERR_CAUGHT);
    }
    FINALLY {
        bn_free(n);
        bn_free(e);
        bn_free(v);
        bn_free(k);
        ec_free(p);
    }
    return result;
}
```

Listing A.3: ECDSA Signature Generation using OpenSSL

```
static ECDSA_SIG *ecdsa_do_sign(const unsigned char *dgst, int dgst_len,
                const BIGNUM *in_kinv, const BIGNUM *in_r, EC_KEY *eckey)
{
        int     ok = 0;
        BIGNUM *kinv=NULL, *s, *m=NULL,*tmp=NULL,*order=NULL;
        const BIGNUM *ckinv;
        BN_CTX      *ctx = NULL;
        const EC_GROUP   *group;
        ECDSA_SIG  *ret;
        ECDSA_DATA *ecdsa;
        const BIGNUM *priv_key;

        ecdsa    = ecdsa_check(eckey);
        group    = EC_KEY_get0_group(eckey);
        priv_key = EC_KEY_get0_private_key(eckey);

        if (group == NULL || priv_key == NULL || ecdsa == NULL)
        {
                ECDSAerr(ECDSA_F_ECDSA_DO_SIGN, ERR_R_PASSED_NULL_PARAMETER);
                return NULL;
        }

        ret = ECDSA_SIG_new();
        if (!ret)
        {
                ECDSAerr(ECDSA_F_ECDSA_DO_SIGN, ERR_R_MALLOC_FAILURE);
                return NULL;
        }
        s = ret->s;

        if ((ctx = BN_CTX_new()) == NULL || (order = BN_new()) == NULL ||
                (tmp = BN_new()) == NULL || (m = BN_new()) == NULL)
        {
                ECDSAerr(ECDSA_F_ECDSA_DO_SIGN, ERR_R_MALLOC_FAILURE);
                goto err;
        }

        if (!EC_GROUP_get_order(group, order, ctx))
        {
                ECDSAerr(ECDSA_F_ECDSA_DO_SIGN, ERR_R_EC_LIB);
                goto err;
        }
        if (dgst_len > BN_num_bytes(order))
```

```
{
        ECDSAerr(ECDSA_F_ECDSA_DO_SIGN,
                ECDSA_R_DATA_TOO_LARGE_FOR_KEY_SIZE);
        goto err;
}

if (!BN_bin2bn(dgst, dgst_len, m))
{
        ECDSAerr(ECDSA_F_ECDSA_DO_SIGN, ERR_R_BN_LIB);
        goto err;
}
do
{
        if (in_kinv == NULL || in_r == NULL)
        {
                if (!ECDSA_sign_setup(eckey, ctx, &kinv, &ret->r))
                {
                        ECDSAerr(ECDSA_F_ECDSA_DO_SIGN,ERR_R_ECDSA_LI
                        goto err;
                }
                ckinv = kinv;
        }
        else
        {
                ckinv  = in_kinv;
                if (BN_copy(ret->r, in_r) == NULL)
                {
                        ECDSAerr(ECDSA_F_ECDSA_DO_SIGN, ERR_R_MALLOC_
                        goto err;
                }
        }

        if (!BN_mod_mul(tmp, priv_key, ret->r, order, ctx))
        {
                ECDSAerr(ECDSA_F_ECDSA_DO_SIGN, ERR_R_BN_LIB);
                goto err;
        }
        if (!BN_mod_add_quick(s, tmp, m, order))
        {
                ECDSAerr(ECDSA_F_ECDSA_DO_SIGN, ERR_R_BN_LIB);
                goto err;
        }
        if (!BN_mod_mul(s, s, ckinv, order, ctx))
        {
```

```
                        ECDSAerr(ECDSA_F_ECDSA_DO_SIGN, ERR_R_BN_LIB);
                        goto err;
                }
        }
        while (BN_is_zero(s));

        ok = 1;
err:
        if (!ok)
        {
                ECDSA_SIG_free(ret);
                ret = NULL;
        }
        if (ctx)
                BN_CTX_free(ctx);
        if (m)
                BN_clear_free(m);
        if (tmp)
                BN_clear_free(tmp);
        if (order)
                BN_free(order);
        if (kinv)
                BN_clear_free(kinv);
        return ret;
}
```

Listing A.4: ECDSA Signature Verification using OpenSSL

```
static int ecdsa_do_verify(const unsigned char *dgst, int dgst_len,
                const ECDSA_SIG *sig, EC_KEY *eckey)
{
        int ret = -1;
        BN_CTX   *ctx;
        BIGNUM   *order, *u1, *u2, *m, *X;
        EC_POINT *point = NULL;
        const EC_GROUP *group;
        const EC_POINT *pub_key;

        /* check input values */
        if (eckey == NULL || (group = EC_KEY_get0_group(eckey)) == NULL ||
            (pub_key = EC_KEY_get0_public_key(eckey)) == NULL || sig == NULL)
        {
                ECDSAerr(ECDSA_F_ECDSA_DO_VERIFY, ECDSA_R_MISSING_PARAMETERS);
                return -1;
        }
```

```
ctx = BN_CTX_new();
if (!ctx)
{
        ECDSAerr(ECDSA_F_ECDSA_DO_VERIFY, ERR_R_MALLOC_FAILURE);
        return -1;
}
BN_CTX_start(ctx);
order = BN_CTX_get(ctx);
u1    = BN_CTX_get(ctx);
u2    = BN_CTX_get(ctx);
m     = BN_CTX_get(ctx);
X     = BN_CTX_get(ctx);
if (!X)
{
        ECDSAerr(ECDSA_F_ECDSA_DO_VERIFY, ERR_R_BN_LIB);
        goto err;
}

if (!EC_GROUP_get_order(group, order, ctx))
{
        ECDSAerr(ECDSA_F_ECDSA_DO_VERIFY, ERR_R_EC_LIB);
        goto err;
}

if (BN_is_zero(sig->r)           || BN_is_negative(sig->r) ||
    BN_ucmp(sig->r, order) >= 0 || BN_is_zero(sig->s)   ||
    BN_is_negative(sig->s)       || BN_ucmp(sig->s, order) >= 0)
{
        ECDSAerr(ECDSA_F_ECDSA_DO_VERIFY, ECDSA_R_BAD_SIGNATURE);
        ret = 0;        /* signature is invalid */
        goto err;
}
/* calculate tmp1 = inv(S) mod order */
if (!BN_mod_inverse(u2, sig->s, order, ctx))
{
        ECDSAerr(ECDSA_F_ECDSA_DO_VERIFY, ERR_R_BN_LIB);
        goto err;
}
/* digest -> m */
if (!BN_bin2bn(dgst, dgst_len, m))
{
        ECDSAerr(ECDSA_F_ECDSA_DO_VERIFY, ERR_R_BN_LIB);
        goto err;
```

```
}
/* u1 = m * tmp mod order */
if (!BN_mod_mul(u1, m, u2, order, ctx))
{
        ECDSAerr(ECDSA_F_ECDSA_DO_VERIFY, ERR_R_BN_LIB);
        goto err;
}
/* u2 = r * w mod q */
if (!BN_mod_mul(u2, sig->r, u2, order, ctx))
{
        ECDSAerr(ECDSA_F_ECDSA_DO_VERIFY, ERR_R_BN_LIB);
        goto err;
}

if ((point = EC_POINT_new(group)) == NULL)
{
        ECDSAerr(ECDSA_F_ECDSA_DO_VERIFY, ERR_R_MALLOC_FAILURE);
        goto err;
}
if (!EC_POINT_mul(group, point, u1, pub_key, u2, ctx))
{
        ECDSAerr(ECDSA_F_ECDSA_DO_VERIFY, ERR_R_EC_LIB);
        goto err;
}
if (EC_METHOD_get_field_type(EC_GROUP_method_of(group)) == NID_X9_62
{
        if (!EC_POINT_get_affine_coordinates_GFp(group,
                point, X, NULL, ctx))
        {
                ECDSAerr(ECDSA_F_ECDSA_DO_VERIFY, ERR_R_EC_LIB);
                goto err;
        }
}
else /* NID_X9_62_characteristic_two_field */
{
        if (!EC_POINT_get_affine_coordinates_GF2m(group,
                point, X, NULL, ctx))
        {
                ECDSAerr(ECDSA_F_ECDSA_DO_VERIFY, ERR_R_EC_LIB);
                goto err;
        }
}

if (!BN_nnmod(u1, X, order, ctx))
```

```
        {
                ECDSAerr(ECDSA_F_ECDSA_DO_VERIFY, ERR_R_BN_LIB);
                goto err;
        }
        /* if the signature is correct u1 is equal to sig->r */
        ret = (BN_ucmp(u1, sig->r) == 0);
err:
        BN_CTX_end(ctx);
        BN_CTX_free(ctx);
        if (point)
                EC_POINT_free(point);
        return ret;
}
```