8-2013

A Secure Reconfigurable System-On-Programmable-Chip Computer System

William Herbert Collins
University of Tennessee - Knoxville, wcollin4@utk.edu

To the Graduate Council:

I am submitting herewith a thesis written by William Herbert Collins entitled "A Secure Reconfigurable System-On-Programmable-Chip Computer System." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Gregory D. Peterson, Major Professor

We have read this thesis and recommend its acceptance:

Syed Islam, Nathanael Paul

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

A Secure Reconfigurable System-On-Programmable-Chip Computer System

A Thesis Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

William Herbert Collins
August 2013

DEDICATION

This work is in dedication to Leon and James Collins, who first explained the difference between parallel and series circuits to me when I was a child and always fostered my exploration of electricity by many discussions and demonstrations, in which we always took pleasure.

ACKNOWLEDGEMENTS

I would like to thank the following people for their contributions: Dr. Gregory Peterson for his advice and insights for this project, without which this project would not have been possible, Dr. Nathanael Paul for improving the cryptographic system, and layout artist Todd Schwarzkopf for the printed circuit board artwork.

ABSTRACT

A System-on-Programmable-Chip (SoPC) architecture is designed to meet two goals: to provide a role-based secure computing environment and to allow for user reconfiguration. To accomplish this, a secure root of trust is derived from a fixed architectural subsystem, known as the Security Controller. It additionally provides a dynamically configurable single point of access between applications developed by users and the objects those applications use. The platform provides a model for secrecy such that physical recovery of any one component in isolation does not compromise the system. Dual-factor authentication is used to verify users. A model is also provided for tamper reaction. Secure boot, encrypted instruction, data, and Field Programmable Gate Array (FPGA) configuration are also explored.

The system hardware is realized using Altera Avalon SoPC with a NIOS II processor and custom hardware acting as the Security Controller and a second NIOS II acting as the subject application configuration. A DE2 development kit from Altera hosting a Cyclone II FPGA is used along with a Secure Digital (SD) card and a custom printed circuit board (PCB) containing a second Cyclone II to demonstrate the system.

User applications were successfully run on the system which demonstrated the secure boot process, system tamper reaction, dynamic role-based access to the security objects, dual-factor authentication, and the execution of encrypted code by the subject processor. Simulations provided detailed examinations of the system execution. Actual tests were conducted on the physical hardware successfully.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER 1
PURPOSE AND ORGANIZATION

1.1 Motivation

Computer systems rely on correct execution of applications to provide data and services that are used in every aspect of life. Financial transactions, health records, mapping services using the Global Positioning System, entertainment systems, and real-time interpretation of data for control systems are a few of the many pervasive digital services that are relied upon in daily life. However, correct execution of code to provide correct answers is not adequate in any scenario in which a person or group wishes to intercept, alter, or use that data in ways not intended or desired by the creator.

Many systems to obscure, or encipher, plaintext data exist as well as their complementary systems to decipher that information. About such methods, Auguste Kerckhoffs wrote:

“The cipher method must not be required to be secret, and it must be able to fall into the hands of the adversary without inconvenience” [1].

The purpose of this thesis is to extend Kerckhoffs’s principle to hardware. Is it possible to create a deployable embedded hardware system such that loss of parts of the system does not compromise the secrets therein? Is it possible to have the architecture of the system be public but to still maintain security?

The first application of such a system that readily comes to mind is a military application. Loss of mobile, embedded hardware to the enemy can mean loss of secret mission data and algorithms, provided the adversaries have resources to recover the data. Additionally, these resources must be protected from internal misuse due to unauthorized access, be it malicious or not.

As a second avenue of exploration, the ability of hardware to be securely reconfigured is explored. In embedded systems, such as in the military example, it may be needed to reconfigure the hardware as the mission progress. It may not be feasible in terms of power or cost to use separate hardware for each aspect of the mission. Can this ability to reconfigure be controlled securely in the same manner as the data and processes?

The goal of this thesis is to explore these topics with modern cryptographic principles as a backdrop.

1.2 Scope of the Thesis

This thesis examines a secure, reconfigurable computing system whose individual physical components may be intercepted by an adversary after it is configured and deployed. This computing system is qualitatively examined in terms of modern cryptographic principles. This computing system serves as a model or prototype in that simplifications are made to make the system more general and manageable in terms of complexity. Cryptographic algorithms are reduced to template functions, as there are many methods to achieve the cryptographic goals. Physical embodiment of tamper detection and response are simplified to a simple user input instead of an actual armoring of the system. The goal of the thesis is to explore cryptographic, reconfigurable, and trust concepts in breadth. The complexity of this goal precludes further exploration into specific cryptographic methods and physically tamper-proof hardware.

The proposed system is realized in hardware and software and tests are performed. Results from those tests are presented along with system simulations that allow for detailed inspection of the system.

1.3 Organization of the Thesis

Chapter 2 provides the reader with a survey of principles needed to achieve the goals of this thesis. Modern cryptographic principles are reviewed, which factor into later design concepts. Reconfiguration of hardware is reviewed. The current state of security concerns is explored along with solutions. The solutions proposed by this thesis are compared to those.

Chapter 3 describes the goals of the hardware system, both cryptographically, and in terms of hardware. The adversary is defined.

Chapter 4 describes the design of a hardware system that will meet the goals outlined.

Chapter 5 describes the design of software systems running on the hardware shown in Chapter 4.

Chapter 6 shows the results of testing the proposed system.

Chapter 7 provides a summary of the findings and explores how to further the work presented in this thesis.

CHAPTER 2
SYSTEM PRINCIPLES AND CURRENT WORK

2.1 Principles of Cryptology

In order to later design a basic cryptosystem for use in this thesis, it is necessary to first define basic cryptographic concepts. These concepts will be used wholly or in part to create a conceptual cryptographic framework for the proposed system.

Cryptology is the “scientific study of techniques for securing digital information, transactions, and distributed computing” [1]. It is comprised of cryptography, which is the study of enciphering messages to protect their content from adversaries [2], and cryptanalysis, which is the study of deciphering ciphertexts without having knowledge of the key used to make them [3].

Claude Shannon developed a model for this interaction between cryptology and cryptanalysis. Shannon’s Model describes a message sender, a message receiver, and an adversary [4]. A plaintext is the original message. The ciphertext is the result of transforming the plaintext by some encryption method [5]. The sender enciphers the desired message using a key and the message. The receiver deciphers the message using the enciphered message and the key [6]. In this model, the adversary is assumed to be able to intercept the message and to have knowledge of the method of enciphering it, but not have the key [6]. In this model, Shannon stipulated two assumptions: (1) the key must be chosen randomly from among all possible keys and all keys are equally likely, and (2) the adversary understands the cryptosystem but does not have the key [6]. This is Kerckhoffs’s principle restated: “The cipher method must not be required to be secret, and it must be able to fall into the hands of the enemy without inconvenience” [1].

Shannon proposed that the encryption method be based on some known problem that is difficult and that the system should be made secure against all known attacks [4].

Such an arrangement is known as a cryptosystem and it encompasses the encryption method, decryption method, plaintexts, ciphertexts and key texts. The following sections examine some important details of cryptosystems [7].

In addition to enciphering and deciphering for secrecy, several other concepts are integral to cryptography and extend the basic model just presented. Authentication, identity, and access control are integral to system security but are

often not first considered in association with cryptology. These concepts, along with the dual of cryptography—cryptanalysis, are explored.

2.1.1 Privacy / Secrecy

The concept that first comes to mind concerning cryptology is the concept of privacy, or secrecy. Suppose two people, Alice and Bob, wish to communicate secretly. In Shannon's model, suppose Alice is sending a message to Bob and Eve is listening. Alice and Bob wish to keep the message private, or secret, so they employ a method to encipher the message under some shared key. The principles behind this scheme are examined in this section.

First, what does it mean for the message to be secret? An encryption scheme is "secure" if "no adversary can compute any function of the plaintext from the ciphertext" [1].

Such perfect security is achievable given some very specific conditions. Perfectly secret encryption is achieved in a system such as the Vernam One-Time Pad (OTP) [4]. In this system, each bit of the plaintext message is added to a random key string that is shared between Alice and Bob. This system meets the definition of secure defined above. It suffers from the problem of the key needing to be as long as the message, or longer [8].

To create a more practical system, algorithms are used with a shorter key such that Alice and Bob both know the algorithm and the key and are able to encrypt the plaintext message with the key to create a ciphertext message that attempts to meet the definition of security.

To formalize the Shannon Model, Alice and Bob share a key generated by a random function Gen( ) which selects a key randomly from the set of all possible keys [1]. Alice performs a transformation on the plaintext using a function

$$M_{enc} = Enc(M, k)$$

where M_{enc} is the ciphertext, k is the key, M is the plaintext message, and Enc is the algorithm. She then sends M_{enc} to Bob, who finds

$$M = Dec(M_{enc}, k)$$

where M is again the plaintext message, Dec is the decryption algorithm, and k is the same shared key. This key must be agreed upon in secret between Alice and Bob prior to this communication. Eve has access to the message M_{enc}, and $Enc( )$ and $Dec( )$ but not the key, k. This scheme is known as symmetric key encryption because both Alice and Bob use the same shared secret key.

Eve intercepts message M_{enc}. It is desired that the algorithm $Enc()$ alters M in such a way that the definition of secure is met. Because the keys are secret, this is known as private key encryption. In military settings, having to meet in person to share a key is not considered a burden as this takes place naturally in staging areas [1].

In many useful cases, it is important for the key k to be able to be made public to simplify key sharing. In this case, $Enc()$ and $Dec()$ use different keys [9]. Each entity has a public key, which is not secret, and a private key, which only the entity knows. To send a message to an entity, message M_{enc}

$$M_{enc} = Enc(M, pk)$$

is calculated with pk being the known public key of the receiver. This message is sent to the entity who computes the plaintext M using his or her private key:

$$M = Dec(M_{enc}, sk)$$

Knowledge of the public key gives no information about the private key [9]. In this manner, a secret message can be sent to an entity without the entities having to establish a secure channel to share the same secret key. This system is known as an asymmetric cryptosystem.

How is the data partitioned to be processed by the algorithm $Enc()$ or $Dec()$? There are two schemes: the block cipher and the stream cipher.

In the block cipher, the plaintext message M is divided into sections, known as blocks, whose length is defined by the algorithm used [8]. Each of these blocks then undergoes a transformation based on the key and algorithm [8]. The goals of the algorithm are “confusion” and “diffusion” as proposed by Shannon [4]. The property of confusion means “the ciphertext statistics should depend on the plaintext statistics in a manner too complicated to be exploited by the cryptoanalyst” [4]. The property of diffusion means “each digit of the plaintext and each digit of the secret key should influence many digits of the ciphertext” [4]. These concepts are central to cryptography. This process is repeated for each block in the message. A reverse process is used to convert the ciphertext back to the original plaintext.

This process of each block being processed independently is known as the electronic code book mode of operation of the algorithm $Enc()$ [10]. However, this suffers from the fact that any repeating part of the plaintext will appear as a repeating segment of the ciphertext and is not desirable [10]. To prevent this, $Enc()$ can be computed in a different mode known as cipher block chaining (CBC) [10]. In this mode, previous outputs of $Enc()$ factor into the key k used for that block: each plaintext block has the previous ciphertext block added to it

(modulo 2) before the block cipher is done [10]. To get started, the first block must have an agreed upon, random value added to it and this is known as the initialization vector [10]. This method has been shown to be chosen-plaintext secure but not chosen ciphertext secure. An example of a block cipher is the popular Rijndael cryptosystem [11]. It allows for block lengths between 128-bits and 256-bits in multiples of 32-bits [11].

What happens if $Enc()$ is applied to the same data twice, but with different keys, $k1$, $k2$? For a symmetric cipher, it turns out that this system is not as secure as expected, which would be as secure as a key of twice the length. This is due to a cryptanalysis method known as meet-in-the-middle. To increase security, it is necessary to use $Enc()$ three times with $k1$, $k2$, and $k3$ to avoid the meet-in-the-middle attack. Triple DES is DES applied three times [12].

A second type of algorithm is the stream cipher. This is a symmetric cryptosystem that works with individual bits such that the output of any ciphertext bit is dependent only upon the key and the previous plaintext bit [13]. To accomplish this, the key provides an input to a pseudo-random bit stream generator whose output forms a running key [13]. This running key is used along with the previous plaintext bit to get the current ciphertext output bit [13]. This system has the advantages of being simple to implement in hardware and fast [13]. A5/1, which is used in GSM telephone transmissions, is an example of a stream cipher [14]. Interestingly, a block cipher can be turned into a stream cipher by operating it in the output feedback mode [10]. In this manner, outputs from the cipher are fed back as inputs to the cipher [10].

2.1.2 Authentication

The second pillar of cryptology is authentication. How does Bob know that the message that he received was the one intended by Alice—that it has not be tampered with or altered? Perhaps an adversary pretending to be Alice sent it (impersonation attack), or perhaps Alice did send the message but the adversary Eve intercepted it and changed it (substitution attack) or sent it again later (replay attack) [15]. Given a read-only memory, how can one ensure that it has not been tampered with?

The answer to this is authenticated encryption (AE). To understand, the concept of cryptographic hash functions must first be defined. A hash function takes inputs of arbitrary length and transforms (hashes, digests) them to create an output of fixed length [16]. A keyed hash function improves upon this by adding a cryptographic aspect such that the output hash depends upon the input message and a secret key. A message authentication code, MAC, algorithm is an example of this concept. It consists of three parts: a random key generation algorithm, a MAC generation algorithm that takes the message and the key and computes the

MAC value, and a MAC verification algorithm that takes the message in question and the key and performs the MAC generation function again and compares the result with the MAC value provided [17]. To work, it must be hard to forge a MAC value on a new text [17].

Authenticated encryption can now be explained. Bob can ensure that the message that he received was from Alice and has not been tampered with by the following method [18]:

1. Alice and Bob share secret keys k_1 and k_2.
2. Alice makes a message M.
3. Alice encrypts M under key k_1 using some cryptosystem to produce M_{enc}.
4. Alice computes the MAC generation S of M_{enc} using M_{enc} and k_2.
5. Alice sends the pair (M_{enc}, S) to Bob.
6. Bob computes the MAC generation of M_{enc} using k_2 and checks that it matches S.
7. Bob decrypts M_{enc} with k_1.

Now Bob is assured of the integrity of the message, of the identity of the sender, and of the secrecy of the transmission in as much as can be guaranteed by the cryptosystem.

Now, if Eve intercepts the message for Bob and replaces it, when Bob computes the MAC verification for the message using the shared secret key with Alice, the value will not match the MAC value presented on the message, as Eve does not have the key required to generate the correct MAC value. Although useful, this method has several problems:

- The MAC has problems in multi-user settings. The MAC is not publically verifiable and transferable—each verifier must have the secret key, so just because Bob verifies his received MAC value, he is not assured that all other receivers will also come to that answer [1].
- The MAC does not allow for the condition of non-repudiation [1]. The sender and the receiver could later disagree about k_2 without any way of proving that they shared the k_2 key for that message [1].

This process is adapted for public-key cryptography [1]. It works as follows:

1. Alice generates an asymmetric key-pair and publishes the public key.
2. Alice “signs” the message by a signing algorithm that takes the message and Alice’s private key and produces a single value known as the signature.
3. Alice appends the signature to the message and sends it to Bob.

4. Bob verifies the signature by using a verification algorithm that takes the message, Alice's public key, and the signature and outputs a true if the signature is valid for that message and false if it is not.

Notice that the message in this case is not encrypted, so this scheme forms only a signature. How can this be added to an encryption scheme to achieve a system similar to encrypt-then-MAC?

A first guess would be to encrypt-then-sign. This has a problem in multi-user settings as described by [19]. Suppose that Alice wants to send a message to Bob. Alice encrypts the message with Bob's public key, and then makes a cryptographic hash of the encrypted portion using her private key. Now she appends this to the message and sends it to Bob. Now suppose that Charlie, who is also a user of the system, intercepts the message. He strips the signature off, runs the encrypted part through his signing algorithm, and then forwards it to Bob claiming the message was from him, even though he does not even know what is in the message. Bob cannot confirm that this did not occur. One solution is for the encrypted part to contain text that enumerates the sender and the receiver. However, if there is only one sender and one receiver, then encrypt-then-sign is secure [19]. Multiusers should, when encrypting data, include the identification of the sender. When signing data, they should include the identity of the receiver with the signed message. Message receivers should check the identity of the sender and the receiver and if it does not match what is expected, reject the message [19]. This thesis uses encrypt-then-sign with only one potential sender, so it is secure under the two-person model.

2.1.3 Identity

In the cryptographic model presented, how can Bob ensure that he is in communication with Alice? Alice is referred to as the claimant as she is alleging to be Alice and Bob is the verifier as he engages Alice in a protocol to ensure her identity [20].

An identity verification protocol is a scheme used to provide entity authentication between parties [21]. It can be unilateral or mutual [21]. Entity authentication is a process by which the claimant proves her identity to the verifier [22]. This can be based on something the claimant knows, something the claimant possesses, or something inherent in the claimant [22]. Passwords are an example of something known, a smart card is an example of something possessed, and biometrics are an example of something inherent [22]. This proof is known as credentials [22].

One straightforward protocol that Bob can employ is the challenge-response identification protocol. In this protocol, Bob presents Alice with a question that changes over time, and she answers the question to prove her identity [20]. This

relies on Alice and Bob having a shared secret prior to this protocol. One simple way to do this is for Bob to send Alice a number encrypted under their shared key, and Alice finds this number by decrypting the message. She then increments the number by one and sends it back to Bob after encrypting it again with the key. Bob decrypts the second message and confirms that the number he originally sent was correctly incremented. In this manner, if Eve attempts to record a message in transient and replay it later, Bob immediately detects it. Bob can also send a sequence number expecting the next value, or even the current time [20].

Perhaps Alice gave her key to a fourth entity. How can Bob ensure that he is in communication with Alice given this possibility? Without a physical tie to Alice, it is impossible to establish this fact. Biometrics address this problem by making credentials based on physical traits or behaviors [23].

2.1.4 Access Control

The fourth aspect of security is the concept of access control. In a system with shared resources, it is important to specify which entities have access to which resources and to further specify the nature of that access. This allows the system to meet security goals such as confidentiality, system integrity, and to prevent certain attacks, such as denial of service [24]. Access control can be defined as a "security function that protects shared resources against unauthorized access" [24]. In the language of security, those resources are called objects and the entities accessing, or attempting to access them, are called subjects [24].

For any system that is to employ access control, it is necessary to have a method to describe how subjects are allowed access to objects. This is known as the access control policy and it can take several forms depending on the goals of the system [24]. The form of the description is known as the access control model [24]. The way in which this policy is expressed determines how flexible the system is to future changes and how easy it is for one subject to delegate rights to another subject [24]. Additionally, the type of model used may make proving security concepts easier as well as making the inspection of the policies more expedient [24].

One type of model is known as the matrix model [24]. In this model, the policy is described by a matrix. Subjects are listed in the rows and objects are listed in columns [24]. Each entry represents the permission that subject has with that object [24]. When using models, one key goal is to "prove safety" which means to determine if access rights are granted to unauthorized subjects [24].

An object-centric listing can be obtained by use of an access control List which is a list of pairs for each security object: $(s, (r_1, \ldots r_n))$ which lists each subject

and the rights of that subject on the object for which the pair refers [24]. This can awkward for systems where subjects can delegate rights to other subjects [24].

A subject-centric listing can be obtained in a similar manner by use of the concept of a capability [24]. A capability is a pair $(o, (r_1 \ldots r_n))$ which lists each object in a system and the rights that the subject to whom the pair refers has [24]. This is also a type of credential [24].

More specifically, a credential is “a token issued by an authority that expresses a certain privilege of its bearer” [24]. One advantage is that some credentials can be verified off-line [24].

Besides the matrix model, another way to express the access control policy is the role-based access model [24]. In this representation, all subjects are assigned to roles and then the roles have rights to objects assigned to them [24]. By this indirection, easier policy management is possible as it is possible to see by inspection what rights a subject has. In addition, this reduces entry errors [24]. For example, in military settings, there might be the roles of unclassified, confidential, secret, and top secret. Subjects could be assigned to these roles, and inspection of the role label applied assures that the correct rights are established for the user, if the roles and their objects are associated correctly.

Finally, the access control policy can be expressed in terms of the information flow model [24]. Under this system, information is tagged with security labels. Subjects can read from lower levels, read and write to their own level, and only write to higher levels [24]. In this manner, information can only flow to equal or more privileged subjects. This is used in need-to-know scenarios such as military applications.

Having an access control policy that can be represented in a detailed, understandable, and mathematically representable way is only the first precondition for having access control. The second is an enforcement mechanism [24]. This entity is located between the subjects and the security objects and has the ability to allow or disallow access [24]. To do this, it is comprised of a part that can inhibit access and a part that determines if the requested access complies with the access control policy [24]. This second part is known as the decision function [24]. In order to prevent a compromise of security, this part must provide complete mediation—it must intercept all accesses to the objects [24]. This is easier to accomplish in centralized systems [24]. Together, these components form an authorization architecture [25].

The concept of access control is central to the hardware design of the system described in this thesis and is revisited frequently.

2.1.5 Cryptanalysis

Cryptanalysis is the process of deciphering an encrypted message without having the key [3]. One must consider the strength of the adversary attempting to do this in order to form a threat model [26]. Who are the adversaries, what access do they have to the different parts of the system and what are their resources? [26]. This will determine the success of the attempted cryptanalysis for a given system.

2.1.6 Methods of Attack

The following is a brief description of attacks that are used to gain information about a plaintext that has been encrypted to form a ciphertext and it is assumed that the key is not known.

Exhaustive key search: the attacker tries every possible key for $Dec(M_{enc}, k)$ to find an expected plaintext value [27]. To resist, the cryptographer should use long keys, change keys often, eliminate known plaintexts, and use cipher block chaining [27].

Dictionary attack: the adversary enciphers an expected plaintext with every possible key and stores this exhaustive list as a look up table [28].

Denial of service: the adversary makes repeated requests to a system such that it does not have time to respond to legitimate access [29].

Code book attack: The adversary collects plaintext-ciphertext pairs for analysis of future messages. This can be avoided with cipher block chaining [30].

Replay attack: The adversary records a communication and plays it back later [15]. Authentication and challenge-response protocols can be used to against this.

Man-in-the-middle: A person intercepts a message and then can alter it, replay it, or learn its contents [31]. Authenticated encryption can be used against this method.

Impersonation attack: an adversary attempts to assume the identity of a legitimate entity in a system [32]. Authenticated encryption and identity verification protocols are used to prevent this.

Chosen ciphertext, or adaptively chosen ciphertext attack: the attacker has the ability to choose a ciphertext and perform the decryption algorithm on it using the

secret key. He does not know the key. In the adaptive version, his choice of new ciphertexts can come from previous trials [33].

Chosen plaintext, or adaptively chosen plaintext attack: the attacker can choose a plaintext to be encrypted using the secret key, but does not have the key [34]. In the adaptive version, new choices for the plaintext are based on previous trials.

Adaptively chosen plaintext and ciphertext: the attacker can do both chosen ciphertext and chosen plaintext at the same time and use the results for future trials [35].

Linear cryptanalysis: the attacker "studies probabilistic linear relations between parity bits of the plaintext, the ciphertext, and the secret key" [36].

Differential cryptanalysis: a general technique for studying symmetric cryptographic systems and focuses on the differences as they evolve through a cipher system [37].

Actual physical attacks can be characterized by who attempts them and what access they have. One classification is clever outsiders, knowledgeable insiders, and funded organizations [26]. The number of systems available for destructive testing is also important along with cost of tools and time necessary [26]. Here is a list of physical attacks:

Environmental attack: cooling the system, heating the system, applying unexpected inputs to the system, or altering the system's time functions are used to attempt to learn secrets from the system [26].

Electromagnetic / power attack: observation of electromagnetic radiation or the power used by the system can help the attacker infer information about the system [38].

Fault injection: an electrical fault can be injected into the system to get it to a state not intended by the designer [39].

Data remanence: some computer memories retain state if cooled, even if power is removed [40].

Software attacks: these attempt to exploit the system security by software means: buffer overflow, unexpected inputs, misinterpretations of inputs, using data at a time much after it was checked, abusing privileges, and accessing undocumented functions [26].

The cryptosystem presented in this thesis will be referenced against this list of attacks to enumerate which ones are considered and modeled and which are not, in order to define the bounds of what can be expected from the proposed system.

2.2 Reconfigurable Hardware

Another goal of this thesis is to explore reconfiguration as part of a secure embedded system. To accomplish this, an FPGA that is under control of the end user is part of the system.

FPGAs accomplish custom functions via use of reconfigurable interconnect to a set array of fixed hardware elements. Take, for example, the Altera Cyclone II. It has a large array of Logic Elements (LEs) that are small combinatorial logic blocks that are optimized to perform synthesized logic functions [41]. These consist of a Look-Up-Table (LUT) that is a function generator of four inputs, a programmable register, inputs and outputs to row and column interconnects, and various ancillary logic [41].

LEs are connected to each other and other on chip devices via a programmable interconnect to realize the system as synthesized, placed, and routed by the Quartus [42] tool. This interconnect, as with all programmable aspects of the Cyclone II, is configured based on SRAM memory cells [41]. These cells are loaded with the desired function during FPGA programming. Programming is accomplished by serial input to predefined pins and must be done on each power cycle.

The Cyclone II also has configurable embedded memory blocks, PLL blocks, multipliers, and Input-Output Elements (IOEs).

The Cyclone III LS provides advanced security features relevant to this thesis. It allows a 256-bit volatile key to be stored internally [43]. This key is used to decrypt the configuration file for the device. The configuration file is encrypted with AES-256. Tamper detection zeros the key. The key is maintained between power cycles by a battery, similar to the concepts in the IBM 4758.

2.3 The Security Problem and Current Solutions

Having reviewed a basic cryptographic model that includes secrecy, authentication, and access control and having reviewed principles of reconfigurable hardware, it is possible to define the current state of security

problems and to show examples of systems that are designed to solve those problems. Then, the design in this thesis can be compared to that work.

2.3.1 The Security Problem

Computer systems process information that is valuable to the user and must be protected. In addition, this information may come to the user from a second party, or may go to a second party; in either case, it is necessary to be able to ensure the source of the information and to ensure that it has not been tampered with. Further, a system may need to attest to another system that it is in a known-good state and to convince itself of that fact before operating fully. Computer viruses, including malware, can infect systems and compromise these goals. In addition, an entity may physically attack a system to acquire the system's valuable information or to change it to operate in an abnormal manner. The same problems apply to smaller embedded computer systems. For example, consider the case of downloadable content for mobile phones. How can the user know that the application cannot access data that the user wishes to keep private? How can a content provider know that a user platform is not duplicating data that was trusted to not be duplicated? Conversely, how can a user know that transactions he or she wishes to perform on a remote machine are occurring correctly and that the machine is in an untampered state? As embedded systems continue to increase in power and run applications that only a full computer could run a few years ago, the need to extend security developed for full platforms to embedded systems will be needed. Consider finally the case of military hardware that could fall to an adversary. It should not provide secrets even it is physically compromised and should not be electronically repurposed for unauthorized use.

To solve these problems, multiple hardware and software solutions are employed. This section covers ones of specific interest to this thesis.

2.3.2 The Security Problem: Access Control

Section 2.1.4 covers the basic principles of access control. Of interest to this thesis is role-based access control (RBAC). The concept of a role-based access control dates to the 1970s in multi-user, multi-application systems [44]. A general-purpose RBAC was generalized by [45]. They note three guiding rules: first, a subject can only execute a transaction if the subject has been assigned a role (or selected one) [45]. Secondly, a subject's active role must be authorized for the subject [45]. Thirdly, a transaction may only be done by a subject if authorized by the subject's role [45]. Note that roles are different from groups in that a group is a collection of users where a role is a collection of permissions. [46]. Role-based access control can also have advanced descriptions that allow for mutual exclusion so that entities cannot have overlapping roles [44].

Additionally, roles can inherit from parent roles [44]. RBAC can be applied to applications to control their access to application data instead of embedding specific code in specific locations to control that access [45]. This can be done in two ways: (1) roles are assigned with permissions to execute entire programs, or (2), roles are assigned with permissions to execute specific functions [47]. RBAC has been studied extensively since the 1990s and was standardized in a 2004 NIST standard based on [46].

Beyond the abstract concept of RBAC, general access control is commonly used to prevent execution in one context from accessing memory regions allocated to another context [48]. Hypervisors are designed to support memory separation between guest operating systems and between the hypervisor and the guests [48]. The Hypervisor Xen is an example of this [48]. The actual mechanism of separation with Xen is very complex, with multi-level page tables where each table can have 1024 entries [48].

Memory can be monitored for unusual operation in the scheme of intrusion detection, then the system could react, and this forms a type of access control. Proposed in [49] is a scheme whereby the code compiler identifies what areas of memory could be changed by the code and under what conditions, and then conveys that to the hardware for enforcement. This is called dynamic access control [49].

General access control is a necessary precondition for establishing a trusted path [50]. Suppose that a user is to enter a password and to interact with a screen that shows banking information. It is necessary to be able to isolate systems to ensure that a trusted path can be made between the program and the peripheral—in this case a keyboard and a screen. Additionally, take the case of a modern motherboard. A CPU connects to memory through a north bridge and to slower I/O through a south bridge. Peripherals can access memory directly through direct-memory-access (DMA). For example, [51] describes the case of a network card accessing memory intended only for the graphics processor. A compromised driver can change the memory mapped I/O (MMIO) region of a device to overlap another device so that it can intercept data [50]. For example, a display could be sent to a network card [50]. Hypervisor design is suggested that provides isolation and prevents these attacks [50].

An ARM processor with TrustZone technology is designed for integrated access control throughout its processor, bus system, and peripherals [52]. The design goal is to provide total separation between a trusted execution context and an untrusted execution context [52]. Objects in the unsecured context cannot access objects in the secured context [52]. Two extra bits are added to read and write signals on the system bus to indicate which context the system is operating in and memory has an extra bit added to it to indicate which context is in use [52]. In addition, some processors have extensions that allow one core to serve in

both contexts in a time-sliced manner [52]. Careful attention is paid to objects that are used for debugging in order to control their access [52].

2.3.3 The Security Problem: Secrecy

Both data and code may be made secret. A platform may execute code stored in a secure area [26]. The encrypted code may be stored in a public location and decrypted prior to execution by the processor [26]. Alternatively, only key parts of the code may be encrypted [26]. The problem of code encryption is difficult [26]. This is because the code should be enciphered with a stream cipher or block cipher operating in cipher block chaining mode as the message space for processor code is small and patterns would easily appear in the ciphertext. However, branches in the code and re-execution of segments of the code make synchronizing the blocks of the running key difficult. Data can be stored as encrypted using standard encryption methods.

2.3.4 The Security Problem: Authentication

The terms “secure boot” [53], “trusted computing,” and “trusted platform module” occur often in computer security literature. These all relate to the concept of trust. “Why should Alice trust computation on Bob’s machine?” [26]. This question is an extension of the cryptographic concepts of secrecy and authentication as it applies to computation. In addition, how can a computer system detect changes to itself?

The answer to this is to create *trust*. Trust is defined by the Trusted Computer Group as “An entity can be trusted if it always behaves in the expected manner for the intended purpose.” Computers could earn our trust by being able to “strongly identify themselves” and to “strongly identify their configuration” [54]. The first part comes from using asymmetric cryptography with the secret key “strongly tied” to the platform and the second part comes from making cryptographic hashes of the state of the machine [54].

To accomplish those two goals, a Trusted Platform Module (TPM) can be added to the system. This module, with specifications defined by the Trusted Computer Group, and the system, provide three roots-of trust and a method to keep track of the state of the system:

1. Root of Trust for Measurement (RTM) is a computation engine that makes reliable integrity measurements [55]. This is usually done by the system’s normal computation engine [55].
2. Root of Trust for Storage (RTS) is a computation engine that maintains digests of the state of the system [55]. This is stored in the TPM [55].

3. Root of Trust for Reporting (RTR) is a computation engine that can report the contents of the RTS reliably [55]. This is integrated into the TPM [55].
4. Platform Configuration Registers (PCR) are a set of registers that are cleared on reset. When written to, they take the current value and combine it with the new value [54].

How can these elements be used to establish trust in a computer? The process of a secure boot provides a solution. In this process, a “chain-of-trust” is built with its roots in the hardware TPM [54]. The chain-of-trust describes the integrity of the boot process and program loading.

What does a TPM measure to keep track of the state of the system so that it can attest to it? Although code execution changes the state of a platform, and attempting to record these dynamic changes could be a goal, the best means of proving the system state is to do a cryptographic hash of code before it executes and to do this for every piece of code that loads during the boot process and to track these through a secure boot process [56]. This process was first described by [53]. This works by always checking code before it is executed. This could be done using the TPM in the following manner: on reset of a system, the PCR is cleared. A measurement is made of the first code segment using RTM. This value is checked against a list that has been signed by an authority (the authority’s public key must reside in the system to do this) [56]. If it is acceptable, then that code segment is loaded and executed [56]. It then makes a hash of the next piece of code that is to be loaded and adds this to the PCR, which takes it and combines it with the previous hash [56]. If an attempt is made to load unauthorized code, then the system halts [56]. The current value of the PCR can then attest to a chain of certified software loads. By making it through the boot process, the system has demonstrated that it is running in a trusted manner [56].

TPMs can also “seal” data by a process of withholding the decryption keys for a memory block or similar until the PCRs are in a specific state. In that manner, the data cannot be read until the system is in an exact state as required by the designers [54]. Microsoft’s BitLocker seals the keys used to encrypt the hard drive and to authenticate it. They are sealed to the state of the PCR.

To accomplish all of these goals, a TPM has key generation ability based on a true random number generator [54]. It also has a non-volatile monotonic counter that helps to prevent against replay attacks [54]. The TPM must be able to demonstrate its identity and to maintain secrecy of keys and this is accomplished with the RTS and the RTR.

How does this concept of secure boot and state measurement apply to reconfigurable FPGA hardware? In [55] a system is proposed that consists of an FPGA that has three sections: a static section that interfaces to passive TPM functions and checks reconfiguration streams, a fixed section that is user-defined

to interface with external hardware, and a dynamically reconfigurable section of the FPGA that is located in the fixed section [55]. Outside the FPGA is a trusted area called the Trust Block that has an initialization ROM configuration of the FPGA along with passive TPM style registers. The ROM initializes the FPGA to describe the static sections and to clear the fixed area [55]. The static area evaluates and checks the user's configuration data for compliance before allowing it to load [55]. Also, PCR registers are used to track the state of the FPGA with special consideration given to reconfiguration [55]. Unlike normal PCRs, these PCRs reset when the system loads a new configuration as no trace of the previous is left on the system [55]. They also propose a system that integrates the TPM functionality completely into the FPGA and assumes an FPGA that can be partially reconfigured [57]. They assume a user application on the FPGA such as a microprocessor and cite the benefits of not having the TPM be a separate entity. Trust is likewise derived from examination of the FPGA bitstream. Special consideration is given to how to maintain secrecy of the TPM information.

2.3.5 The Security Problem: The IBM 4758 Solution

The IBM 4758 secure co-processor is one example solution to the security problems outlined and provides an excellent example of a complete system designed with security inherently.

The goals of the IBM 4758 encompass those defined previously and expand on them:

1. To provide physical armor to prevent tampering by an adversary [26].
2. To protect designated data from the adversary given the fact that he has access to part of the system. Protection of data means ensuring that the data is secret and authentic [26].
3. To prove code is executing in a correct environment [26]. In this manner, multiple parties who have differing motivations can participate in a computation securely [26].
4. To possibly provide secrecy of the code executing on the system [26].
5. To be general purpose [26].
6. To support multiple operating systems [26].
7. To support multiple vendors [26].
8. To allow for security in the presence of mutually untrusted parties [26].
9. To be reconfigurable [26].
10. To be easy to update [26].
11. To allow for updates to the cryptographic engine [26].
12. To ensure secrets go away when under attack [26].
13. To ensure secrets start out secret [26]. This means careful consideration of the deployment of the system from factory initialization to end use.

14. To ensure secrets remain secret even in the presence of a software attack [26]. This is a result of number (8) as code running on the system may have been designed with different motivations.

It uses the concept of secure boot to defend against software attacks [26]. As with the general system that is improved with the addition of a TPM, the IBM 4758, upon reset, executes code from an initial, trusted space that cannot be altered [26]. This code then certifies the next segment of code that is to be executed and passes control to it [26]. The first section of code cannot be reached again without resetting the device. The second segment of code executes, checks the integrity of the third layer of code, and passes execution to it. Likewise, this second segment of code cannot be reached again without a system reset, which causes the entire process to start over [26]. The access control policy of the platform uses the boot stage to decide what accesses are allowed [26]. To ensure that the system cannot be tricked by software into operating as if it were in a lower, more privileged stage, a hardware counter, called the trust ratchet, is incremented at each step and cannot be rolled back without a system reset [26].

Concerning physical attacks, tamper response refers to how a system responds to a detected tamper event. The IBM 4758 system employs tamper reaction in the form of sensor circuits that could detect a tamper event and a system that zero secrets and that takes the system off-line [26]. These tamper events include physical penetration, alteration of expected electrical inputs, and changes in temperature [26].

Beyond the mechanics of the device itself, the deployment model of a system is a key factor to the system's design and overall security as well as complexity. How is the device initialized at the factory [26]? How is it transported to the user [26]? How is code maintenance done [26]? Is it possible to reuse the device for another purpose [26]? Can the system be audited [26]? How is software developed for it [26]? Can the device be exported [26]? Is the device secure enough for the environment [26]? Is it feasible and robust [26]? What levels of compromise should be assumed [26]? Will it need updates [26]? These factors were carefully considered in the design and deployment of the IBM 4758.

Many of these design parameters can be qualified by proving that the system meets the FIPS 140-2 standard. FIPS 140-2 [58] is a standard that defines what it means for a security device to be secure. The IBM 4758 is a level four system and has the following characteristics: it resists the highest level of physical attacks, all cryptographic functions are performed in a protected envelope, and it is intended to be used in physically unprotected areas [58].

The IBM 4764 represents a later entry into the market. The IBM 4764 is a PCI-X based secure co-processor that performs high-speed cryptographic functions and

is also certified at FIPS 140-2 Level 4 [59]. It has an IBM PowerPC microprocessor [59]. It also has hardware for DES, Triple DES, AES, hashing, and public-private key algorithms [59]. A secure clock and a hardware random number generator are also provided [59]. Like the IBM 4758, it clears its secrets on detection of a tamper event [59]. Typical applications are anywhere high-speed data encryption and signing are needed that may be traced to trustworthy hardware [59].

2.3.6 The Security Problem: A Role-Based SoPC Reconfigurable Solution

The purpose of this thesis is to design a system that could be used as a template for a secure, reconfigurable computing platform. At a high level, such a system is comprised of two parts: one part that controls all security aspects of the system and is not user alterable, and a second part that is a user area, which can run user applications, both hardware and software. Additionally, this user area could be reconfigured in hardware while in use. This system uses cryptographic principles to make the system secure. The system employs a role-based access model that would make it well suited for military applications. The system divides secrets between hardware components such that loss of part of the system would not provide information about the data stored on that part. It offers a dual-authentication system to authenticate the user.

This thesis proposes a system that draws on the principles and previous solutions described and adds to them. A system is proposed that has the following features:

- Hardware-enforced RBAC: the system will determine access control based on the precepts of RBAC. The access control is flattened so that it can be described in a simple matrix of subjects vs. objects. Inheritance and mutual exclusion can be allowed for by direct description in this simple matrix; in other words, this will not automatically occur. This access control ensures that security objects (peripherals) are adequately isolated from each other and trusted paths. The hardware enforcement will be address-based so that it can control access to a device but does not perform semantic inspection of data that being transacted. General access control to the entire platform is provided by a dual-authentication system of a password and a key-card.
- Reconfiguration: The system will support reconfigurable hardware. Descriptions of the hardware will be on a key-card and will be encrypted. The platform will authenticate and decrypt the configurations before loading them. A reconfigurable area of an FPGA is assumed while maintaining an immutable area.

- Context switching: As part of reconfiguration, the system will allow commands to be received that will instruct it to reconfigure in a secure manner using data from the key card.
- Computation: The system will use a soft-core processor as an example of a user-defined computer system.
- Secrecy: The system will be designed with consideration of physical recovery by an adversary of different parts. Data will be encrypted so that recovery of individual parts does not provide useful information. The processor can directly encrypt and decrypt instructions in a simplified ECB model, and can do the same with data.
- Trust: The system will provide a simple model to attest its trustworthiness to the user application.
- System on Programmable Chip (SoPC): The system will be designed using SoPC technology.
- Tamper reaction: the system will model a tamper event with a push-button and will clear its secrets.
- Authentication: configuration data signatures are checked before loaded. The user will enter a password and a key card to use the system.
- User-initiated data secrecy: A unique system whereby each user has the ability to selectively encrypt and decrypt data with keys that are not related to the board itself is added.
- Deployment: A detailed deployment model and threat model will be presented.

Chapter 3 defines these goals specifically.

CHAPTER 3
DESIGN

This section defines the threat model that the proposed system, when fully implemented, is designed to resist. Specifications of the system are also given.

3.1 Threat Model

To understand the system, it is necessary to understand the threats that it is designed to resist. The thesis system has simplifications to make implementation more manageable. The threat model assumes that a full embodiment of the system is being attacked.

The proposed system is designed around the following types of attackers:

- Funded organizations who can uncover any secrets hard-coded in the silicon. They cannot overcome the tamper-detection and zeroing system.
- Knowledgeable insiders who develop user software applications. Although this system allows the user to develop hardware and software, the system is only designed to resist user software attacks. User hardware attacks are not considered and present an interesting area of study.

The proposed system assumes the following time-line:

1. The hardware is in a safe place when the user accesses it and inserts the key card. The hardware remains in the safe place during the check of the key card and the loading of the user application.
2. The key card is removed and the system is deployed to the field. In this scenario, run-time user reconfiguration is not allowed. Run-time user reconfiguration is only allowed while operating in the safe area as it requires the key card, which is no longer with the system.
3. The system operates in the field and is captured.
4. The system is attacked and tamper-detecting circuits zero the secrets. This could be easily extended to zeroing the entire FPGA, in which no information about the purpose of the FPGA could be found, but that is not implemented in this thesis.
5. All information that remains after a tamper reaction event is available to the attacker.

Two other cases are also considered. First, the key card becomes available to the attacker for whatever reason, and secondly, the key card and the running system are captured at separate times. The estimated system responses to these cases are examined in Table 3.

3.2 Design Specifications

The following two tables enumerate the thesis specifications. Table 1 lists all of the cryptographic concepts reviewed in Chapter 1 in one column, and then lists the specification for that concept. The specification may appear in one of two columns. The first column lists the specification without qualification. The second column lists a “Simplified Design Goal” which is a simplification to make the thesis scope manageable. If a specification is not listed next to a concept, this indicates that it is not in the scope of this project. Table 2 is in the same format and lists the goals of the hardware system as first envisioned along with simplifications as needed.

Table 1. Security Design Goals

Cryptographic or Security Design Concern	**Design Goal**	**Simplified Design Goal**
Cryptosystems	Use standard cryptographic functions	Provide template functions for standard cryptographic functions
Compromise model	Any secret stored on a piece of hardware is encrypted with a key held by the remaining parts of the system.	
Symmetric key system	128-bit block cipher	Provide template functions
Asymmetric key system	Full system that uses symmetric keys with public-private keys to speed decryption	Provide template function, only use public-private keys
Stream cipher		
Electronic code book mode	Avoid ECB	Note where used
Cipher block chaining mode	Use CBC	Note where it should be used
Multiple encryption on data	Allowed as needed	
Authenticated encryption	Encrypt-then-sign	Provide template functions for encrypt-then-sign (valid for a two-party system)
Identification protocol: passwords	Use as part of identification protocol	
Identification protocol: challenge-response	Use challenge-response for any key-fob tokens	Use passive token
Identification protocol: biometrics		
Access control policy: access matrix	Develop an access control policy expressible in a matrix	
Access control policy: role based	Allow for four roles: unclassified, confidential, secret, top secret	Use three roles: level 1, level 2, level 3
Access control policy: information flow		
Complete Mediation	The access mechanism should intercept and control all accesses	
Data is secret	Data stored on system is encrypted	User application can choose what is stored encrypted and to have it encrypted automatically
Code is secret	Code stored on system is encrypted	Select code segments can be encrypted and executed directly given special constraints (ECB mode)
Data is authenticated	Data is cryptographically signed	Provide template functions for Sign() and Verify()
Code is authenticated	Code is cryptographically signed	Provide template functions for Sign() and Verify()
Secrets go away when under attack	Keys are cleared on reset; FPGA is de-configured	Keys are cleared; FPGA configuration is maintained.

Table 1. Continued.

Cryptographic or Security Design Concern	Design Goal	Simplified Design Goal
Secrets start secret	Suggest deployment model	
Secrets remain secret with software attack	User software must not be able to violate the access policy	
Secure Boot	First booted code segment is unalterable, it checks and loads higher level code segments, which do the same	Boot code is unalterable and performs system checks before allowing user access
Trust Ratchet	Use a hardware counter as a Trust Ratchet	
Exhaustive key search		
Dictionary attack		
Denial of service attack		
Code book attack		
Replay attack		
Man-in-the-middle attack		
Impersonation attack		
Chosen-text attacks		
Linear / Differential cryptanalysis		
Environmental attack		Simulate with a hardware input
Electromagnetic attack / electrical attack		
Physical armor		
Fault injection attack		
Data remanence attack		
Software attack	System resists user software attacks	
System is general purpose		
System supports multiple operating systems		
System supports multiple vendors		
Mutually suspicious software components	System resists user software attacks	
Reconfigurable	User system is reconfigurable	
Easy to update		
Deployment model	Describe the build process for the system, including encryption	
FIPS-140-2 Level		Similar to Level 4 in concept
Random, uniform keys	Select keys randomly from a uniform key space	Keys manually selected

Table 2. Hardware Design Goals

Hardware Concept	**Design Goal**	**Simplified Design Goal**
Platform Microprocessor	Commonly used microprocessor	NIOS II soft-core processor without interrupt support
Microprocessor Clock	200 MHz	50 MHz
User token	Active	Passive SD card
System peripherals	UART, SDRAM, PIO, Terminal	PIO, Terminal, Internal RAM
Two component solution: dual FPGAs	One FPGA provides security functions, the second FPGA provides reconfigurable USER functions	One FPGA demonstrates security and user functions, Second FPGA demonstrates secure reconfiguration only

CHAPTER 4
HARDWARE DESIGN

4.1 System Architecture Design

This chapter describes the hardware design of a system that meets the goals presented in Chapter 3. Explored in this section are the high-level hardware design, the high level FPGA design, and the security design. Built upon this is the software system that is presented in the next chapter.

4.1.1 Hardware Design

Figure 1 depicts the hardware system as originally designed. This will be referred to as Case 1. Two printed circuit boards (PCBs) and a Secure Digital (SD) card contain all of the necessary hardware. The primary one is the DE2 development kit from Altera. This board will be referred to as the DE2 Main Board.
It has an Altera Cyclone II in addition to many peripherals. The second PCB was designed and fabricated for this thesis and has a Cyclone II and support circuitry along with two seven-segment displays connected to the Cyclone II. This board is designed to plug into the expansion header of the DE2 board. This board will be referred to as the DE2 Sub-Board.

The system consists of two parts: a user part and a security part. The user part is contained on the User FPGA, is designed by the security subject, and is reconfigurable during run-time.

The security part is designed by entities who define the access control policy, referred to as the system owners. Its function cannot be altered by the user during runtime. It provides the sole point of contact between any hardware designed by the user and the system peripherals. In that manner, it meets the definition of complete mediation.

The system deployment model is as follows. The user develops hardware and software to run on the system. This hardware is defined by binary files describing the User FPGA configuration and the applications that may run on the user-defined hardware, such as programs for a NIOS II. These binary files are encrypted and placed on an SD card. The user application design, in order to communicate with the host peripherals, is required to include a component known as the User Bridge. This component provides any encryption services that the user may wish to employ and it performs a serialization of the data bus to reduce the I/O count needed in hardware. The user's only interface to the rest of the system is by connecting to the Security Bridge on the

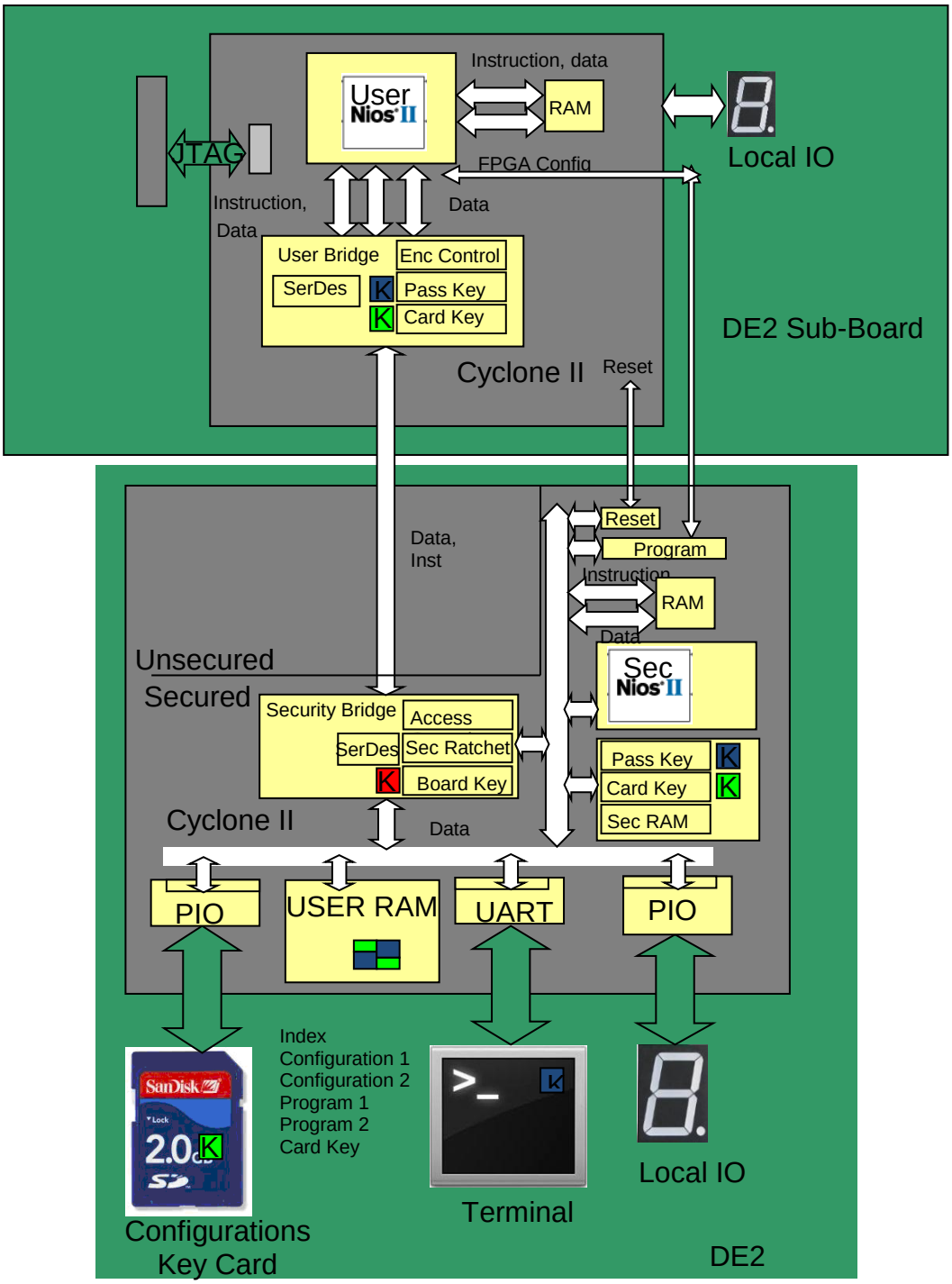

Figure 1. System Hardware, Case 1

DE2 Main Board and it is suggested that this be done with the specified User Bridge.

When the user wishes to use the system, he or she inserts the SD card and enters a password. The User FPGA is configured as designed by the user, and if a NIOS II is present, its code is copied to user RAM for execution.

The security components on the security FPGA handle many services to ensure correct and secure operation of the system. They are commanded by a NIOS II running the security system code. The security components are called the Security Controller.

This system, Case 1, most clearly demarks the secure, fixed parts of the system from the unsecure, user-designed parts. This was the original design. This proved to complicate development too much, although the DE2 Sub-Board is designed to accommodate that approach.

To meet the system design goals and to simply the system, Case 2 is designed and implemented in hardware. It is presented now, as it is a simplification of Case 1. Figure 2 shows how the system is simplified. The user NIOS II is moved to the DE2 Main Board along with its User Bridge and associated components. The DE2 Sub-Board FPGA (User FPGA) is now not connected to the Security Bridge, but its configuration is still under control from the DE2 Main Board. In this manner, the secure control of an FPGA configuration can still be demonstrated. The same deployment model is assumed, but with the realization that this system is an example model as it would not be desirable for the user to use the same development platform that is used for the security control components.

4.1.2 FPGA Design

The FPGA high-level design for this thesis is done using Altera Qsys. This is a system-on-programmable-chip design (SoPC) tool. Figure 3 shows the SoPC system architecture including addresses that the firmware uses to access those components. This is a more detailed view of the internals of the FPGA on the DE2 Main Board for Case 2. Important to note at this level is that the Security Controller NIOS II is the Qsys component nios2_qsys0 and the user Nios II is the Qsys component nios2_qsys1. It can now be seen that all connections from nios2_qsys1, the user NIOS II, are mediated through the avalon_bridge_master_side component, which is the User Bridge, and it connects to the avalon_bridge_slave_side, which is the Security Bridge.

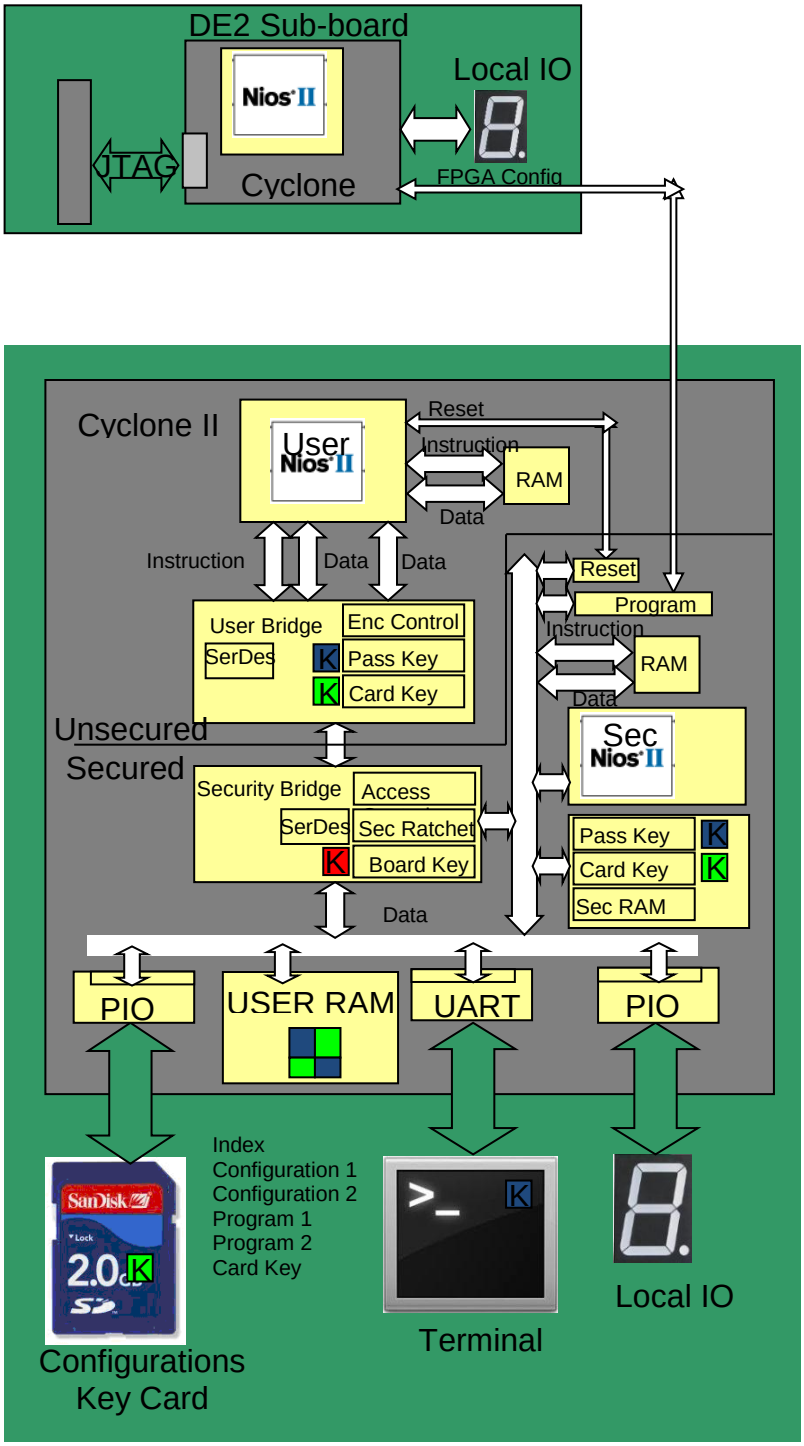

Figure 2. System Hardware, Case 2

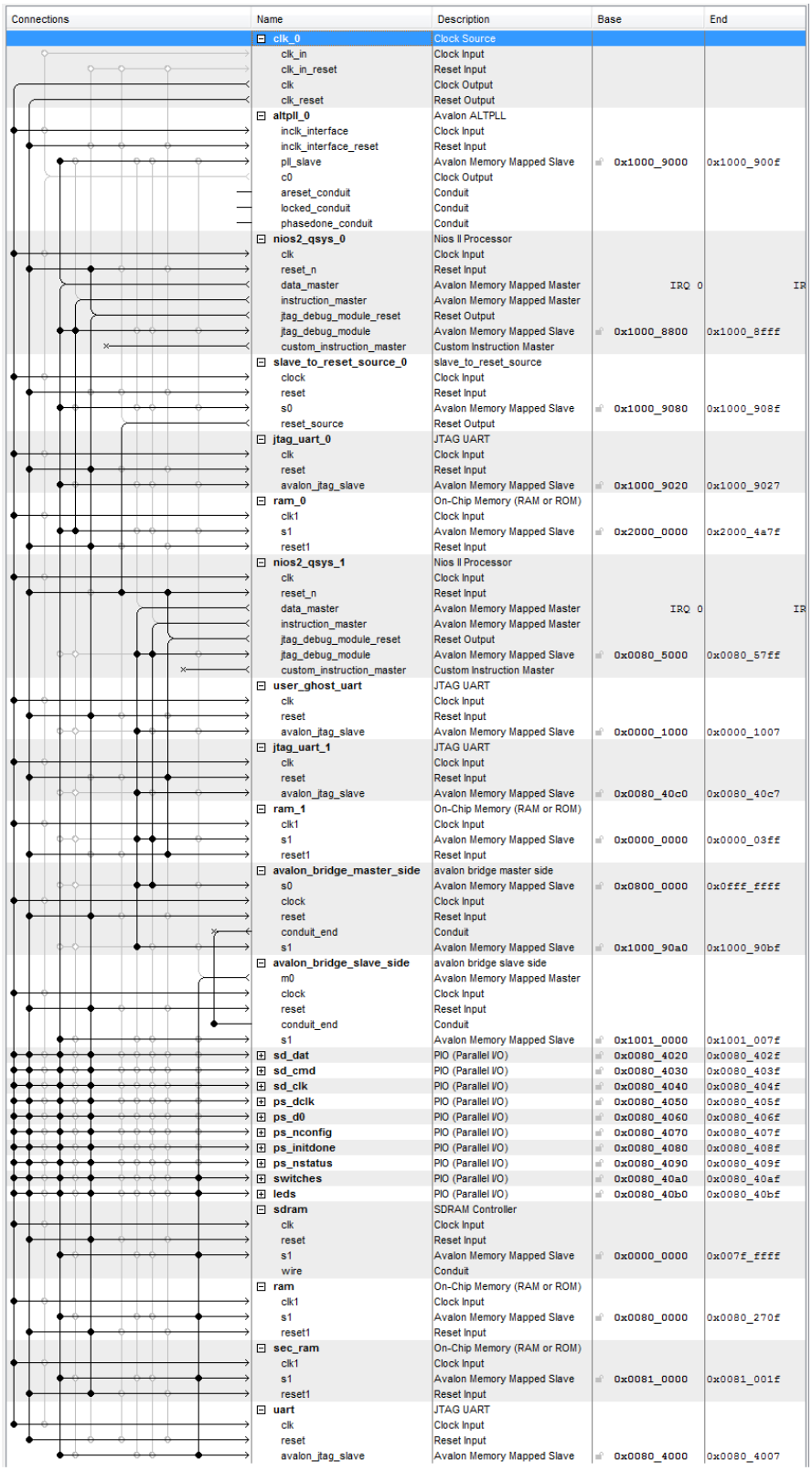

Figure 3. DE2 Main Board FPGA Internals--SoPC

System Contents | Address Map | Clock Settings | Project Settings | Instance Parameters | System Inspector | HDL Example | Generation

	nios2_qsys_0.data_master	nios2_qsys_0.instruction_master	avalon_bridge_slave_side.m0	nios2_qsys_1.data_master	nios2_qsys_1.instruction_master
ram_0.s1	0x2000_0000 - 0x2000_4a7f	0x2000_0000 - 0x2000_4a7f			
avalon_bridge_slave_side.s1	0x1001_0000 - 0x1001_007f				
slave_to_reset_source_0.s0	0x1000_9080 - 0x1000_908f				
jtag_uart_0.avalon_jtag_slave	0x1000_9020 - 0x1000_9027				
altpll_0.pll_slave	0x1000_9000 - 0x1000_900f				
nios2_qsys_0.jtag_debug_module	0x1000_8800 - 0x1000_8fff	0x1000_8800 - 0x1000_8fff			
sec_ram.s1	0x0081_0000 - 0x0081_001f		0x0081_0000 - 0x0081_001f		
leds.s1	0x0080_40b0 - 0x0080_40bf		0x0080_40b0 - 0x0080_40bf		
switches.s1	0x0080_40a0 - 0x0080_40af		0x0080_40a0 - 0x0080_40af		
ps_nstatus.s1	0x0080_4090 - 0x0080_409f				
ps_initdone.s1	0x0080_4080 - 0x0080_408f				
ps_nconfig.s1	0x0080_4070 - 0x0080_407f				
ps_d0.s1	0x0080_4060 - 0x0080_406f				
ps_dclk.s1	0x0080_4050 - 0x0080_405f				
sd_clk.s1	0x0080_4040 - 0x0080_404f				
sd_cmd.s1	0x0080_4030 - 0x0080_403f				
sd_dat.s1	0x0080_4020 - 0x0080_402f				
uart.avalon_jtag_slave	0x0080_4000 - 0x0080_4007		0x0080_4000 - 0x0080_4007		
ram.s1	0x0080_0000 - 0x0080_270f		0x0080_0000 - 0x0080_270f		
sdram.s1	0x0000_0000 - 0x007f_ffff		0x0000_0000 - 0x007f_ffff		
avalon_bridge_master_side.s0				0x0800_0000 - 0x0fff_ffff	0x0800_0000 - 0x0fff_ffff
avalon_bridge_master_side.s1				0x1000_90a0 - 0x1000_90bf	
nios2_qsys_1.jtag_debug_module				0x0080_5000 - 0x0080_57ff	0x0080_5000 - 0x0080_57ff
jtag_uart_1.avalon_jtag_slave				0x0080_40c0 - 0x0080_40c7	
ram_1.s1				0x0000_0000 - 0x0000_03ff	0x0000_0000 - 0x0000_03ff
user_ghost_uart.avalon_jtag_slave				0x0000_1000 - 0x0000_1007	

Figure 4. Address Map

Also, notice that the Security Controller Nios II connects to many components as it has complete control of the system. Figure 4 shows an address map of the system. Each device that is a bus master is listed along with the addresses at which it reaches its peripherals. This is an introduction to the FPGA internals. A more detailed examination of important components is provided in Section 4.2

4.1.3 Secrecy

Secrecy is integral to the hardware design, so the security model used to achieve the design goals is explored in this section and is depicted in Figure 5. The system uses three main keys, known as the board key, the pass key, and the card key, which are referred to as sk_{sc}, k_{pass}, and k_{card} respectively. These are shown in color in Figure 1 and Figure 2 as well.

The pass key, k_{pass}, comes from the user when prompted to enter a password. It is comprised of sixteen hex characters for a total of sixty-four bits. Entry of this password is strictly controlled by the Security Controller. It is critical that a system using this model ensure that the password entry is via a trusted route such that software attacks on it are not possible. Ideally, this would even be a separate key entry area with a dedicated display apart from any keyboard used by the end user. This is a part of a symmetric key pair. The card key, k_{card} comes from the user token--the SD card. It is a sixty-four bit value. Ideally, this would be held in a secure place on the user token, and the token could be verified with a command-response protocol by the Security Controller. For this thesis, the concept is simplified and it is stored on the SD card encrypted.

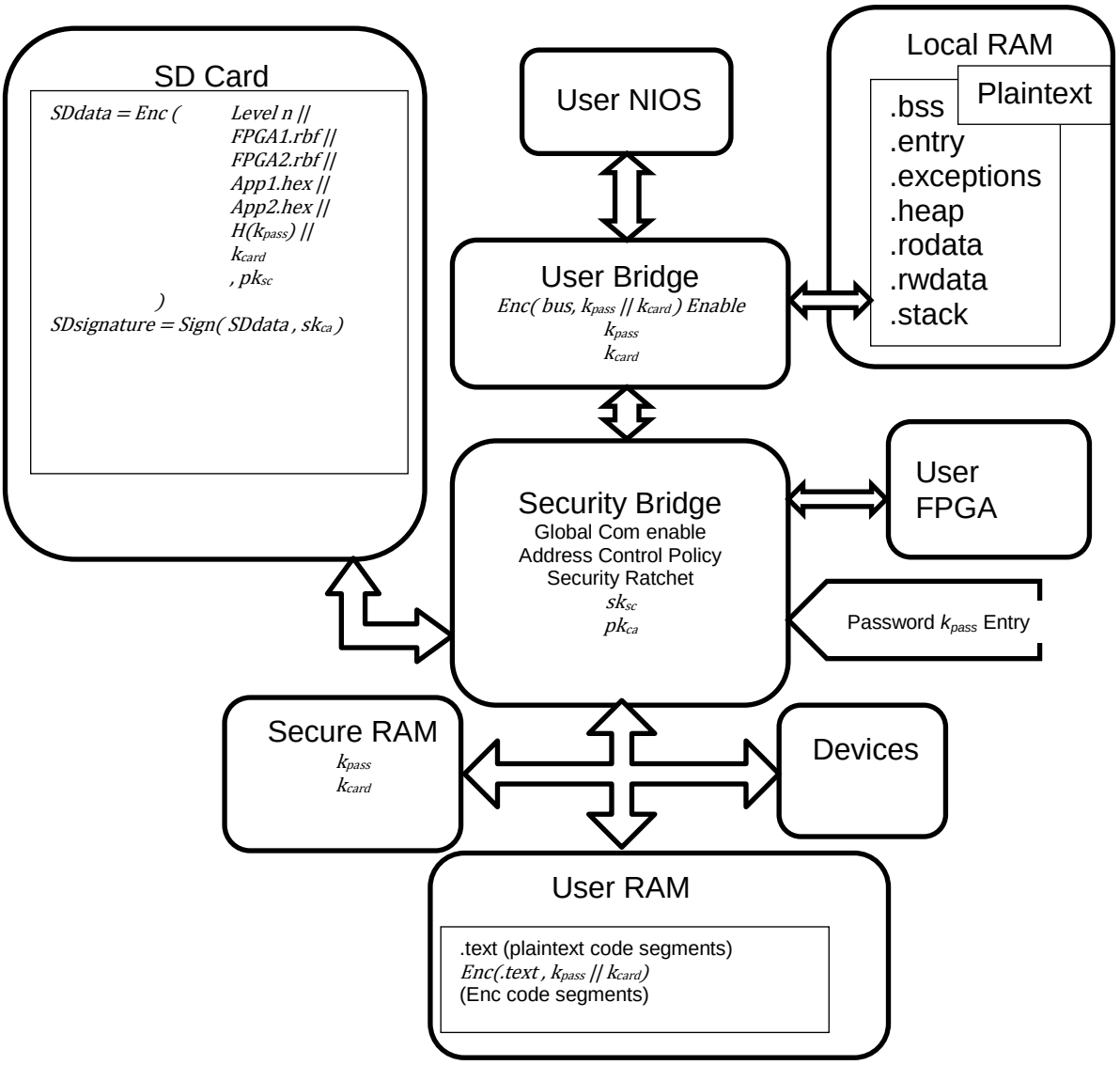

Figure 5. Security Design

K_{card} is concatenated (denoted as ||) with k_{pass} to form a 128-bit key. This key is part of a symmetric pair used by the user for his or her encryption system in the user application. Exclusive-or would normally be preferred as one key does not imply information about the other and the system maintains the same number of bits in the security parameter if part is lost. In this thesis, two 64-bit keys concatenate to 128-bits for demonstration. Ideally, this could be two 128-bit keys or higher mixed by exclusive-or (|| replaced by $\oplus$ everywhere).

The board key, sk_{sc}, is a special key that resides in the FPGA hardware of the Security Controller and is part of the system design and deployment model. It is the private key of a public-private key-pair that enables configuration information to be written securely to the Security Controller.

In the diagram, and hereafter, "*Enc(...)*" refers to an encryption function where the first parameter is the value being encrypted and the second is the key. If listed with a public or private key, then it is assumed to be an encryption taking place in a private-public cryptosystem, otherwise it is assumed to be a symmetric cipher. In this manner, the system may be described generically. The same holds for *"Dec( )"* as a decryption function. *H(...)* is a general collision-resistant hash function that takes an input and digests it to a hash value. *Sign(...)* is a cryptographic public-private key signing algorithm that takes data to sign as the first parameter and a private key as the second parameter and returns a signature value that is the output of a cryptographic hash. *Verify(...)* is the verification function for *Sign(...).* The first parameter is the claimed message to verify, the second parameter is the signature value generated by *Sign(...)*, and the third parameter is the public key associated with *Sign(...)*. It returns true or false to indicate the validity of the signature. These cryptographic functions are simplified and are described in Chapter 5.

The key system can now be explained as follows: all secrets on the SD card are encrypted with the public board key and signed by the private key of the system owners who are acting as the certificate authority CA. As such, if the SD card is physically acquired by an adversary, the secrets on it are held as strongly as the encryption that stores it. One could encrypt a symmetric key in the SD card using the Security Controller's public key, and then encipher the data on the SD card with the symmetric key to improve speed when the Security Controller unpacks it. The model here is simplified to indicate that data is passed to the Security Controller using a public-private key system.

The password key, k_{pass}, is held by the operator who must ensure its secrecy.

The DE2 Main Board and the attached sub-board have two secret areas. One, the user RAM, is implemented as a RAM component in this thesis design, but conceptually it could be a flash memory where the user wishes to store permanent applications or data. This data can be encrypted by the user under

the pass key and card key by the User Bridge or deployed as encrypted. In this way, should the adversary acquire the main board setup, and additionally find the board key on the security FPGA, the data saved on the system RAM / flash again is as secure as the encryption system used to store it, as there is not a trace of the pass key or the card key on that part.

The second secret stored on the DE2 Main Board, as alluded to, is the board key. Finding this key only gives the adversary the board key as the data on the DE2 Main Board is stored under the pass and card keys.

Refer to Table 3 and Table 4 for a full listing of scenarios where entities have portions of the system and the resulting compromises. Note, this system is designed with a 128-bit composite key in mind, but it could be extended to 256-bit to increase security.

The user Nios II has a plaintext area in the tightly coupled RAM adjacent to it. This RAM is part of the user-defined configuration, does not persist between system uses, and allows for the user Nios II to execute faster.

Operationally, the system works as follows:

1. The system owners decide on a role-based access matrix for the system address map. All system peripherals are accessed through a memory map based model. All users are assigned to roles.
2. The system owners generate public-private key pair pk_{ca}, sk_{ca} for signing.
3. The system owners generate a board public-private key pair pk_{sc}, sk_{sc}.
4. The system owners generate a symmetric key k_{pass} for each user.
5. The system owners generate a symmetric key k_{card} for each user.
6. The system owners develop the system hardware and embed sk_{sc} and pk_{ca} into the Security Controller.
7. The system owners place the access control policy in the Security Controller firmware.
8. The system owners secure the Security Controller FPGA. Secure configuration is available on certain devices from FPGA manufactures such as Altera.
9. The system owners give the security subjects a template for the User Bridge for incorporation into their designs and an address map for the system so that the subjects can access the hardware. They are also given k_{pass} and k_{card}.

Table 3. System Compromise when an Adversary Recovers Components

Case	DE2 Main Board System Recovered	User Token Recovered	Password Recovered	Adversary has these Keys:	Adversary has this Data	Strength Bits
1			X	k_{pass}		n †
2		X		$Enc(k_{card} , pk_{sc})$	$Enc(SDdata , pk_{sc})$	l ‡
3		X	X	k_{pass} $Enc(k_{card} , pk_{sc})$	$Enc(SDdata , pk_{sc})$	l
4	X			sk_{sc}	$Enc(userdata , k_{pass} \| \| k_{card})$	n
5	X		X	sk_{sc} k_{pass}	$Enc(userdata , k_{card})$	$n/2$ ††
6	X	X		sk_{sc} k_{card}	$SDdata$ $Enc(userdata , k_{pass})$	$n/2$ ††
7	X	X	X	*All keys*	*All data*	0

† Security parameter for symmetric keys (n-bit system)
‡ Security parameter for asymmetric keys (l-bit)
†† Exclusive-or keys instead of concatenating to make this n.

Table 4. System Compromise Summary by Case

Case	Summary
1	The adversary only has the password k_{pass}.
2	He has the user's configuration encrypted with the board key pk_{sc}. He has physical access to the User Token.
3	He has the user's configuration encrypted with the board key pk_{sc}. He has physical access to the User Token.
4	The adversary has broken the DE2 Main Board. He has unencrypted parts of application flash memory and unencrypted user data (if any—this is user determined) . He has physical access to the DE2 Main Board.
5	The adversary has broken the DE2 Main Board and has the password. He has unencrypted parts of application flash memory and unencrypted user data (if any—this is user determined). He has degraded the encryption of user data and applications for designs using key-concatenation instead of XOR mixing. He has physical access to the DE2 Main Board.
6	The adversary has broken the SD card and has FPGA1 and FPGA2 and the unencrypted parts of application flash memory (if any—user determined) . He has degraded the encryption of user data and applications for designs using key-concatenation instead of XOR mixing. He has physical access to the DE2 Main Board. He has physical access to the User Token.
7	The adversary has the same privileges as the user.

10. The security subjects, hereafter referred to as the users, design and develop system hardware and firmware to meet the goals.
11. The users send the FPGA configuration files and application .hex files to the system owners via some application.
12. The system owners append n, the user's security level, to the data.
13. The system owners append $H(k_{pass})$ to the data.
14. The system owners append k_{card} to the data.
15. The system owners encrypt the data with the public board key pk_{sc} and then sign the data with their private key sk_{ca}. This application could automatically load and format the SD card or active token.
16. The user wishes to use the secure platform to accomplish her goals and therefore is in proximity of the system as this is not a distributed secure system.
17. The user applies power or resets the secure platform.
18. The system resets and secrets that compromise encrypted data on the system are cleared.
19. The system performs security checks.
20. The user enters the SD card when requested.
21. The Security Controller reads the *SDdata* and the *SDsignature*.
22. The Security Controller runs $Verify(SDdata, SDsignature, pk_{ca})$ on the SD card data. It halts if false.
23. The user enters pass key k_{pass}' when requested.
24. The Security Controller checks $H(k_{pass}') = Dec(Enc(H(k_{pass}), pk_{sc}), sk_{sc})$. It halts if false.
25. The Security Controller sets the access policy based on the level $Dec(Enc(n, pk_{sc}), sk_{sc})$.
26. The Security Controller sets $k_{card} = Dec(Enc(k_{card}, pk_{sc}), sk_{sc})$.
27. The Security Controller loads the user FPGA1 configuration. It terminates if there is a problem.
28. The Security Controller clears user RAM and loads the user APP1 program. User-selected secure parts of this are already encrypted by the user and now decrypt to $Enc(App1\ secure\ segment, k_{pass} \,||\, k_{card})$.
29. The Security Controller now enables communication on the Security Bridge.
30. The Security Controller starts the user Nios II.
31. The user Nios II, if the user wishes to use encryption, must copy k_{pass} and k_{card} from the shared security RAM to the User Bridge.
32. The user Nios II executes and uses the services available to it.
33. The user Nios II may request the current hardware configuration of the system which is $H(Security\ Controller\ ROM) + H(hardware)$ where hardware represents any component whose configuration is under control. This is simulated by the switches on the DE2 board.
34. The user Nios II may request the value of the security ratchet.
35. The user application may elect to run $Enc(bus, k_{pass} \,||\, k_{card})$ or $Dec(bus, k_{pass} \,||\, k_{card})$ where bus is the combined instruction/data bus which originates at

the User Bridge and ends at the Security Bridge. This can occur at any time and is controlled by the user.

36. The user application may request a context change wherein FPGA2 and APP2 are loaded and checked in a similar process as steps 26-35. If it is running in the second configuration, it can request a switch back to the first, again, in a process consisting of steps 26-35.
37. Any system reset clears k_{card} and k_{pass} from the User Bridge and from the Security RAM.

4.1.4 Authentication

The system authenticates, to itself, the user by checking a hash of the user's password against the decrypted hash provided on the SD card that was signed by the system owners.

The system authenticates, to itself, the configuration data by checking its signature from the system owners before decrypting it.

The user application authenticates, to itself, the hardware system that it is running on by requesting a hash of the current board state and by checking the security ratchet. The board state hash is a simplification of a PCR. It is a hash of the Security Controller Nios II RAM and switches on the board.

4.1.5 Access Control

The Security Controller enforces a role-based access policy based on its reading of the signed user's access level. There are three access levels, level 1 through level 3. The system owner predetermines what security objects can be accessed for which levels. The Security Controller is designed to allow this matrix to be changed in real time by the Security Controller Nios II microprocessor. In this manner, the system is more flexible.

The access policy is expressed as a matrix in Table 5. Each subject can be dynamically allowed to access three different memory-mapped regions, plus peripherals. The Security Controller performs real time cycle level inspection of data to and from the user Nios II.

Table 5. Access Control Template

User Level	Address Range 1	Address Range 2	Address Range 3	LEDs	Switches	UART
1	Low:high	Low:high	Low:high	Yes / No	Yes / No	Yes / No
2	Low:high	Low:high	Low:high	Yes / No	Yes / No	Yes / No
3	Low:high	Low:high	Low:high	Yes / No	Yes / No	Yes / No

4.2 System Component Design

This section details key design aspects of the components of the system described the in previous sections.

4.2.1 SoPC Design

System on Programmable Chip (SoPC) design refers to the use of reconfigurable hardware (FPGAs) to build System On Chip (SoC) designs [60]. These have a microprocessor, peripherals, memory, and custom hardware designed on a single chip [60]. The configurable FPGA hardware for this thesis is designed at a high level using Qsys, which is an Altera tool that expedites SoPC design by providing a graphical interface to instantiate intellectual property cores and to generate their interconnections [42]. At a lower level, custom hardware modules are developed in Verilog, which is C-like hardware description language that allows various abstraction levels [61].

The complete SoPC system is shown in Figure 3. The system is designed by instantiating and customizing Qsys-provided components and the custom designed components and connecting them. Then the system is compiled and the Qsys tool generates a large amount of quite complex interconnect logic to connect the system. This interconnect is comprised of Altera-designed Avalon components. The Avalon interconnect fabric is an open, high-speed interconnect fabric that is designed for streaming data between hardware cores and for reading and writing to registers and for controlling off-chip devices [62]. These components perform a host of complex functions such as serialization, de-serialization, arbitration, routing, multiplexing, and demultiplexing. The interconnect structures are compliant with the Avalon bus specification and so are the custom hardware modules that are designed. The Avalon interconnect is best described as a small internet-like serial interconnect that achieves high speed by using very wide bus widths such that much data is transacted on each cycle. Data is encoded, routed, arbitrated, and decoded when it moves from point to point. The output of the Qsys tool is a set of Verilog files and a set of test bench files that are imported to Quartus for synthesis, mapping, placing, and

routing. The test bench is imported into ModelSim for system simulation. ModelSim is a widely used graphical simulator for hardware description languages such as Verilog and is made by Mentor Graphics [63]. The Qsys output is placed into a larger Verilog container, which tells the SoPC design how to connect to the physical board. The system in Figure 3 is comprised of the parts detailed in this section.

4.2.2 Avalon SoPC Components

This section is a brief overview of the standard Avalon components used in the system and focus is on the more interesting high-level components. Note: lower case names are system components.

The NIOS II Processor: this is a 32-bit Harvard-architecture RISC processor that can address 2 Gbytes of space [64]. It can be instantiated in three versions. The version used for this thesis is the NIOS II/e with a minimum feature set.

The user sets the reset vector and exception vector. The processor also comes with a joint test action group (JTAG) debug port [65]. The full version of the NIOS II incorporates more advanced features such as a hardware multiplication unit, configurable instruction and data caches, configurable interrupt controller, and configurable memory management support.

There are three total NIOS II/e processors in the system. The Security Controller uses a NIOS II for command and control (nios2_qsys_0). Its data bus is connected to the local RAM, Security Bridge, Security RAM, the user RAM, the SD card via a Parallel Input/Output (PIO) interface [42], a Universal Asynchronous Receiver/Transmitter (UART) [42] for user interaction, a reset controller that controls the user Nios II reset, LEDs, switches, the user UART, and finally, PIO used to program the User FPGA. Its instruction bus is connected to the same local RAM.

The second is the user Nios II (nios2_qsys_1). Its data bus is connected to a local RAM for fast, unencumbered access to data and is also connected to the User Bridge so that it can access the full system. The instruction bus is connected to the local RAM and also to the User Bridge so that it can execute code from user RAM. It has a second connection to the User Bridge that allows it to configure the bridge.

The third Nios II appears on the User FPGA as an example configuration. Refer to Figure 1.

Clock Source (clk_0): this SoPC component takes an actual clock input and redistributes it along with a reset to every component in the system.

Altpll (alt_pll_0): this component is a configurable PLL used to condition the system clock such that it is advanced by 10 ns for use with the SDRAM.

JTAG UART: this is a UART to JTAG interface that allows the system to communicate with a terminal window. There are multiple ones in the system. Jtag_uart_0 is connected to the data bus of the Security Controller NIOS II and is used to communicate with the user and to get the pass key password. Jtag_uart_1 is connected directly to the user NIOS II, is inserted for development only, and would not be used in a full system. Uart is a JTAG UART connected to the Security Controller data bus and can be accessed by the user Nios II according to the security policy. Ghost_uart is a JTAG UART connected to the data bus of the user Nios II and is in place as a software construct. For a system developed as Case 1 in Figure 1, it is necessary for the Eclipse complier board support package to recognize and understand that a UART is present on the data bus. The User Bridge prevents the tool from seeing the UART attached to the Security Controller, so it does not allow for its use. For development purposes, the ghost_uart could be addressed at the same location so that the Eclipse tool can compile for it.

RAM: there are multiple RAMs in the system design. Ram_0 is the Security Controller Nios II RAM and is initialized with the Security Controller program when the Security Controller FPGA is initialized. It is located on the data and instruction buses of the Security Controller Nios II. Ram_1 is the local RAM for the user Nios II. It is located on the instruction and data buses of the user Nios II. It can be initialized according to the user application needs. It can be configured to contain all of the user code, obviating the need for the user RAM that is mediated through the Security Controller. For this thesis, the user application is specified to run in the user RAM through the Security Controller. Ram is located on the Security Controller data bus and is accessible by the user Nios II as allowed by the access policy.

Parallel Input/Output (PIO): The Avalon PIO component allows for general-purpose input and output. The following is a list of PIO controllers that are connected to the Security Controller Nios II data bus: sd_data, sd_command, and sd_clock drive the SD card. Being PIO, the SD interface is in software. Additionally, ps_dclk, ps_d0, ps_nconfig, and ps_nstatus are command and status lines to the User FPGA to configure it and confirm that it is configured. The configuration protocol is designed in software to follow the passive-serial programming option for the User FPGA. Other PIO connections to the Security Controller data bus are switches and LEDs. These interface to the switches and green LEDs on the DE2 board and make them software addressable.

SDRAM: The Avalon component labeled sdram is an Avalon-based SDRAM controller and connects to the SDRAM on the DE2 Board. The SDRAM is not

used in this thesis. Although the hardware functioned, the Avalon intellectual property SDRAM module would not properly process the data for converting its 16-bit contents to a 32-bit word in this system.

4.2.3 Designed SoPC Components

Several system components are designed without building on existing components. Their designs are detailed in this section. They are built upon the Avalon interface specifications by Altera [62].

The Avalon architecture supports several interface types based on the system goals. The “Memory Mapped” interface type is used for custom modules in this thesis as it best serves to connect the custom components to the Nios II bus systems. Components designed as memory-mapped can be either a “memory-mapped slave” or a “memory-mapped master.” An Avalon bus can have multiple masters and multiple slaves. The Qsys tool inserts proper interconnect logic to perform arbitration and routing decisions. The bus structure is flexible in that it has features that are optional when designing components for it.

The Avalon bus masters designed in this thesis connect to the Avalon fabric with a data bus (separate inbound read data and outbound write data), an address bus, a read signal, a write signal, and a wait request. For a master to write data to a location on a slave, the data is placed on the data bus and the address to which it is to be written is placed on the address bus. Next, the write signal is asserted which tells the Avalon fabric to receive data from the master and to route it to the destination address at the slave. The master then waits for the wait request signal to clear, indicating that the data was received. The read process is complementary. The Avalon master places the requested address on the address bus and asserts the read signal. It then waits for the wait request to clear. Then it reads the data from the data bus and uses it as needed.

This thesis also makes use of custom Avalon slave interfaces. These operate in a manner that complements the master. When their read or write signals are asserted, they respond by accepting or posting data to the data bus. While busy, they post a wait request signal. Of particular interest are the User Bridge and Security Bridge, which have both a master and two slaves.

The following paragraphs describe the design of the custom components used in Figure 3:

Slave_to_reset source: this is a custom-designed peripheral that is on the Security Controller Nios II’s data bus. It allows the reset of the User Nios II to be controlled by the Security Controller software. It has one Avalon slave input and it has an output that controls the reset of the User Nios II.

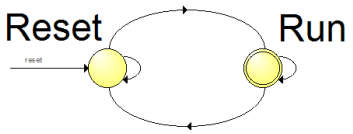

Figure 6. Slave to Reset Source State Machine

The logic used to implement the Avalon slave is a Moore state machine. In state Run, it holds the user Nios II reset as not asserted. In state Reset, it asserts the reset on the user Nios II. The decision to change states is based on the slave port. When it receives a write, if the data is 1, then state Reset is selected. If the data received from the master is 0, then state Run is selected. The state machine states are shown in Figure 6.

The User Bridge: this more complex custom component is shown as avalon_bridge_master_side in Figure 3.

This component performs three functions for the user Nios II. First, it serializes and de-serializes the user Nios II Avalon bus to the Security Bridge, which reduces the number of pins needed for Case 1 in Figure 1 from ninety-five to twenty-five, at the cost of running more cycles. Secondly, it stores a copy of the pass key and the card key on the user side of the system. These locations are cleared on reset. Finally, it provides encryption and decryption services for the connection between the user Nios II and the Security Bridge.

This component is designed to be as transparent to the upstream master as possible. The incoming data is 64-bits wide, (it has separate inbound and outbound data lines that are 32-wide) but are serialized down to two 8-bit buses. The 24-bit address bus passes through without change.

This component has a slave port that is connected to the instruction bus of the user Nios II and the data bus of the user Nios II. It also has a second slave port that connects to the data bus of the user Nios II. It has an Avalon master output that connects to the Security Bridge slave port. The second slave port connection to the user Nios II data bus master is so that the user Nios II can directly configure parts of the User Bridge. The code can reference the card key and the pass key directly from this hardware and the code can enable and disable encryption on the bus.

It functions as follows. Two state machines operate this component. The primary state machine is the main state machine that performs serialization, de-serialization, and encryption on the bus. It is depicted along with state names

in Figure 7. There are two main paths through this machine. The first path through the machine is a write cycle in which the machine takes the incoming write data from the upstream master, which is received on this component's slave port, and serializes it and writes it out on this component's master port, encrypted if needed. The secondary machine (Figure 8) stores the pass key, the card key, and an enable for encryption. There are two flows through this state machine, a read path and a write path, similar to before. In the write path, the user writes the pass key, the card key, or the encryption enable value. In the read path, these values are repeated back to the bus.

The encryption function, as with all encryption functions in this thesis, is a template function. It is implemented as:

$$Enc(\ bus\ ,\ k_{pass}[47:44]\ ||\ k_{card}[47:44]\ ||\ k_{pass}\ [43:40]\ ||\ k_{card}\ [43:40]\ ||\ k_{pass}\ [7:4]\ ||\ k_{card}\ [7:4]\ ||\ k_{pass}\ [3:0]\ ||\ k_{card}\ [3:0]\ )$$

Where $Enc(\ )$ mixes the bus with the pass key and the card key using an exclusive-or function. The decryption function takes the same form. Since this encryption does not depend on prior encryptions, it works in electronic code book mode. This mode, as opposed to cipher block chaining, is used to simply the function.

The Security Bridge: this component is labeled as avalon_bridge_slave_side in Figure 3. This component performs the majority of the security-based functions of this system. Refer to Figure 9 and Figure 10. Its slave connects to the User Bridge bus master on the upstream side, and its master connects to slave ports on all security objects on the downstream side. It has a second slave port that connects to the Security Controller Nios II data bus so that the Security Controller Nios II can configure this component from software. It provides the following functions:

Serialization / de-serialization: this component serializes and de-serializes the bus from the User Bridge and converts it to a normal 32-bit bus for connecting to the security objects.

Access control: this component inspects every address access from the user Nios II on a cycle-by-cycle basis and compares it to the security policy stored in it by the Security Controller Nios II. If the access is not valid, it is ignored and any requested data is zero. It also has a master control that causes all communication to be ignored.

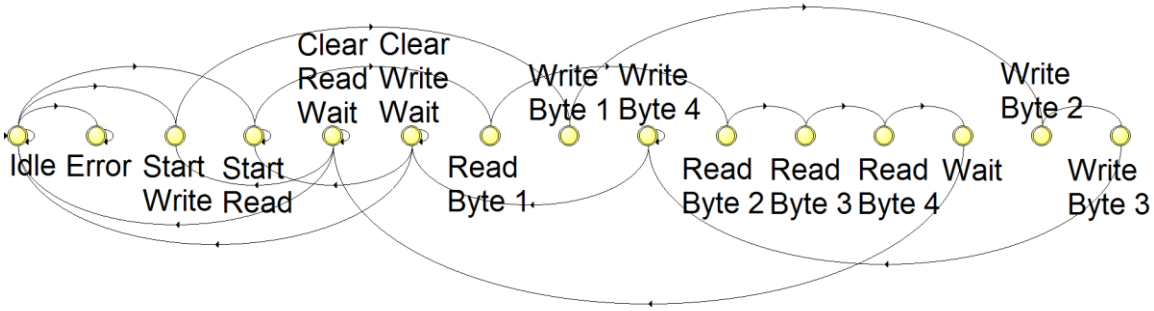

Figure 7. User Bridge Primary State Machine

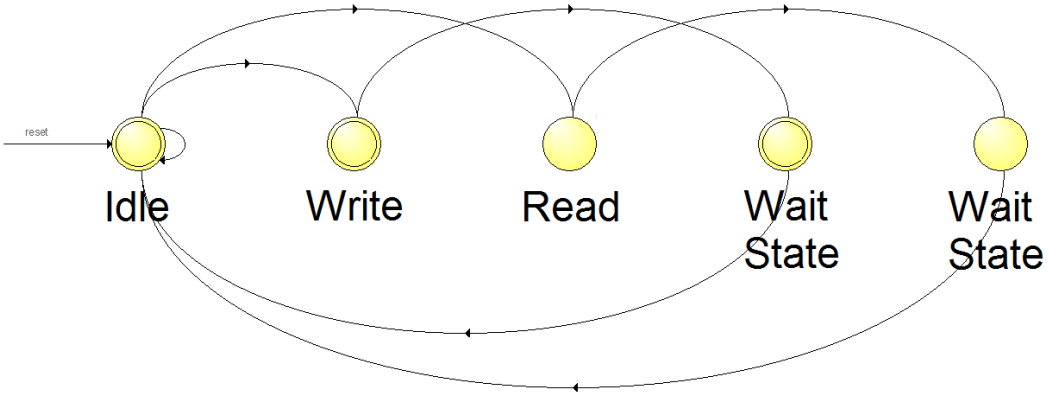

Figure 8. User Bridge Secondary State Machine

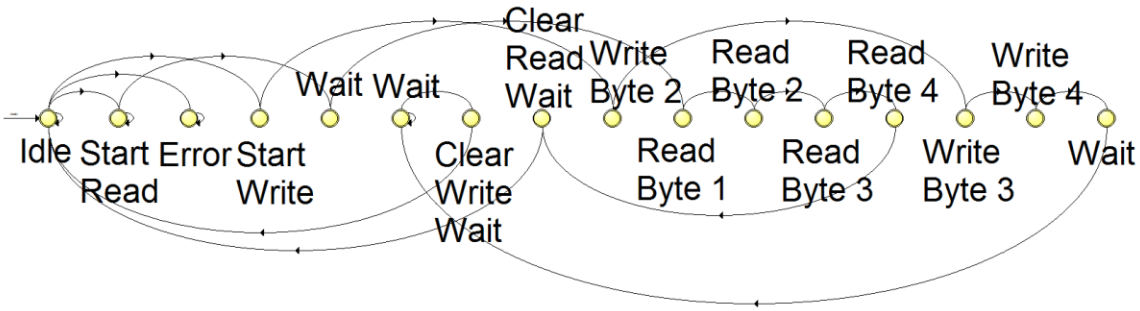

Figure 9. Security Bridge Primary State Machine

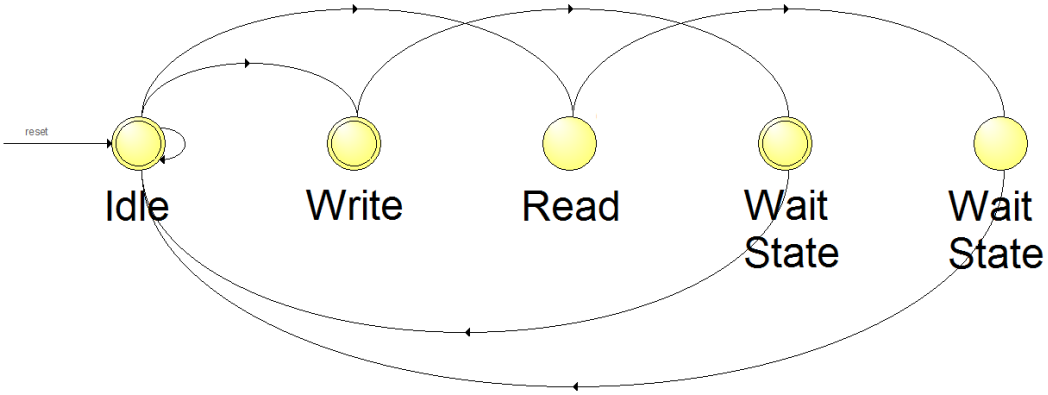

Figure 10. Security Bridge Secondary State Machine

Board Key Storage: the board private key is stored here.

Security ratchet: a security ratchet, based on the IBM 4758, is implemented in hardware. It is a counter that goes from zero to eight. Writing any value to it increments it. Reading it returns its current value. It can only be set back to a lower number through a reset. The two state machines for this component are very similar to the User Bridge and are only summarized. In Figure 9, two flows through the state machine can be seen. One is the read, in which the component gets a read request from User Bridge and it processes it by communicating with the downstream peripherals. The other path through is a write cycle, in which this component receives data from the upstream User Bridge and writes it to the downstream devices.

As with the User Bridge, there is a second slave port on this device that allows it to be configured and the state machine that controls this function is depicted in Figure 10. The slave port provides access to the parameters that set the access policy shown in Table 5. It also provides access to the security ratchet, the board key sk_{sc}, and as access to a global control that inhibits the user Nios II from any communication.

The Security Bridge inspects the incoming address by a large combinatorial logic function that is the core of the access control mechanism. It references a set of registers that define the low and high address ranges and devices that are allowed, as seen in Table 5.

Sec_ram (security RAM) is a 32-byte RAM shared between the user Nios II and the security Nios II and allows them to communicate apart from the Security Bridge. This allows the user Nios II to get the pass key and the card key from the Security Controller, to request a checksum of the current configuration of the board, and to request a context change. The sec_ram component has only one state and takes write data from the bus and stores it, or provides read data to the bus when commanded. It has a custom Verilog implementation that ensures that it clears on reset. This is not true of regular Altera SoPC RAM components.

This section reviews the hardware that runs on the DE2 Main Board security FPGA. This hardware must physically connect from the security FPGA to devices on the DE2 board. The next section briefly details how this is accomplished.

4.2.4 Development Platform

Refer to Figure 11 and Figure 12. The SoPC system just presented is configured in the Security Controller FPGA. For it to connect to hardware on the DE2 board, it is instantiated in a high-level Verilog file that specifies the pin-out so that it can connect to the DE2 board. This file specifies that the SoPC system connects to

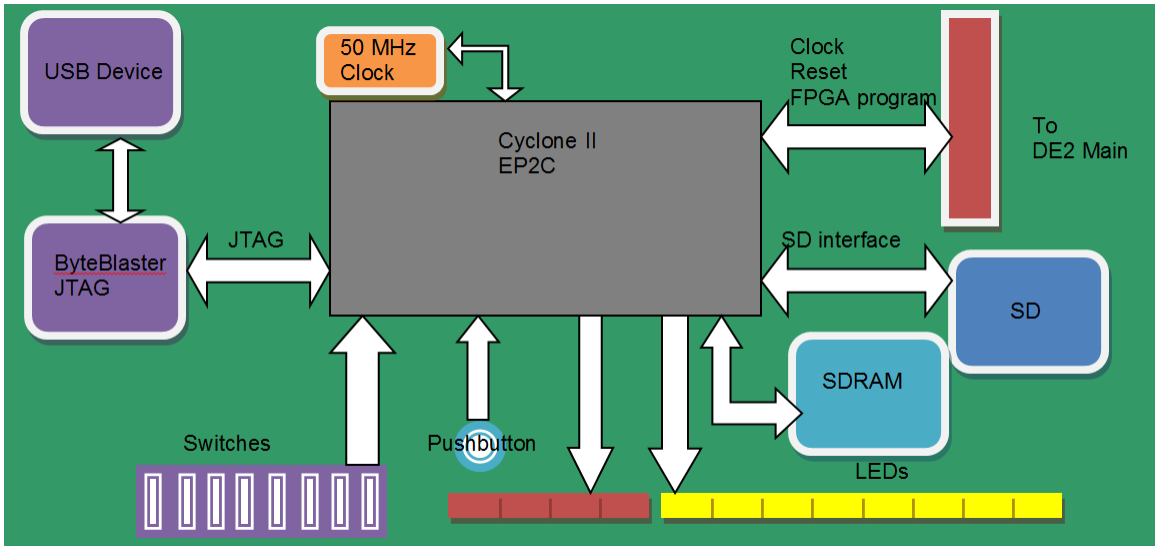

Figure 11. DE2 Main Board

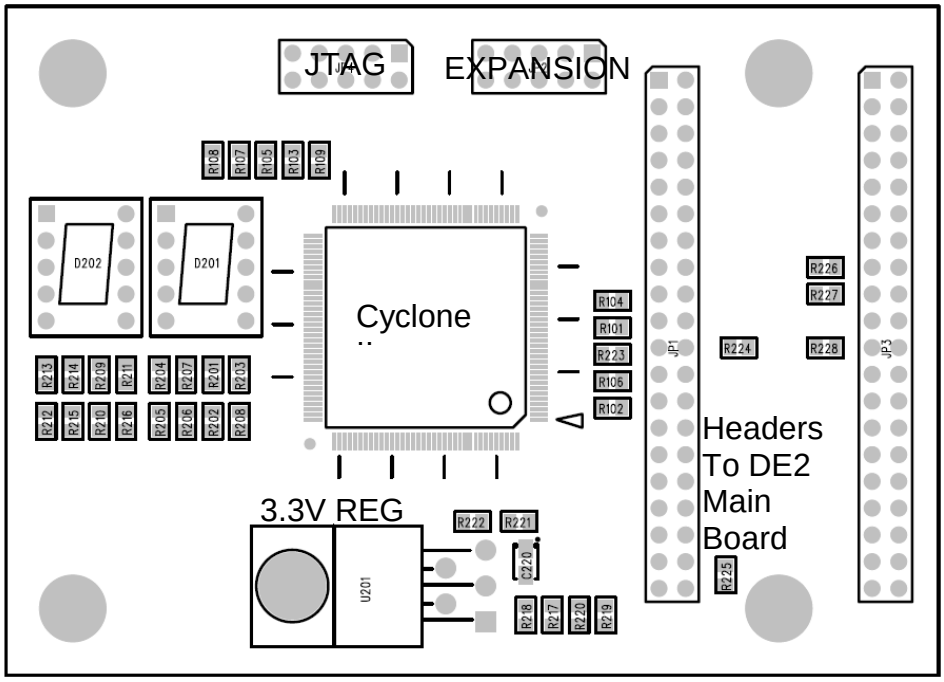

Figure 12. DE2 Sub-Board

the system 50 MHz clock, that push button 0 connects to the system reset, that the switches and LEDs are connected to their respective PIO components, that the SD card connects to the SD card PIO components, that the User FPGA configuration lines connect to their PIO components, and that the SDRAM connects to the SoPC SDRAM controller. Also, at this level, a clock is provided to the DE2 Sub-Board along with a set of red LEDs that show the status of the programming lines to the DE2 Sub-Board. A 50 MHz clock provides timing for the system. The DE2 is programmed through the USB byte blaster.

4.2.5 DE2 Sub-Board Platform

The DE2 Sub-Board plugs into the DE2 Main Board expansion headers. The DE2 manufacturing drawing is depicted in Figure 12.

This board is a custom-designed PCB with a Cyclone II FPGA, 3.3 V regulator, and two seven-segment LED displays. The Cyclone II connects directly to the headers through carefully laid LVDS pairs. It also connects directly to the seven-segment displays. Its clock, configuration, and configuration status pins also go to the header as they connect to the Security Controller FPGA for programming and verification. The PCB is a six-layer board that is designed with good return path and decoupling. The layout was imported to HyperLynx and the extracted layout was analyzed for rise and fall times and termination using a field solver. The FPGA is programmed in passive-serial mode using custom software running in the Security Controller Nios II.

4.2.6 User FPGA Configurations

Two FPGA configurations are designed that demonstrate the user's ability to configure an FPGA from a secure source. The configurations are depicted in Figure 2 as a Nios II in the User FPGA. Both configurations have a basic Nios II processor, JTAG UART, as well as PIO that interfaces to the seven-segment display. For simplicity, the only difference between the two configurations is what is displayed on the seven-segment LEDs. In this manner, it is possible to tell what state the FPGA is configured in.

4.2.7 SD Card

The Secure Digital (SD) card is a 2 Gbyte SanDisk memory card [66]. The FPGA configuration files, the Nios II application files, the card key file, the key hash file, the security level file, and the security header file are stored on it using HxD Hex editor [67]. The Security Controller accesses the files using low level reads and

writes using libraries based on work by [68]. This reduces the code requirements for the Security Controller.

CHAPTER 5
SOFTWARE DESIGN

The software for the Security Controller and the user applications is designed in the C language using the Eclipse tool set with Nios II extensions. The Eclipse toolset provides a full, integrated development environment [69]. The C language, being function-oriented, is ideal for highlighting the hardware as described in Chapter 4. The extra cost of using a more object-centric language such as C++ is avoided, although the toolset allows for this.

The software is designed in C and compiled for the Nios II processor. The Eclipse toolset takes a description of the SoPC platform as described in Chapter 4 and builds a user-customizable board support package (BSP) for it. This includes many convenient input and output functions, user customizable linker tables, and many other options to specifically tailor the BSP for the application needs.

Two complete systems are designed in software. The first is the Security Controller software. This executes on the Nios II in the Security Controller. This processor has complete control of every device in the system. The second are the two user applications. They have their own complete BSP and application files.

5.1 Security Controller Application

The purpose of the Security Controller software is to initialize the system, check the system integrity, enforce the access control policy, and perform the user services of loading and verifying FPGA configurations and application files for the user Nios II.

5.1.1 Board Support Package

In order to build a Nios II based Security Controller, it is necessary to ensure that the entire system can easily be accessed in software and that basic functions for any needed peripherals are present. Figure 13 shows the BSP for the Security Controller.

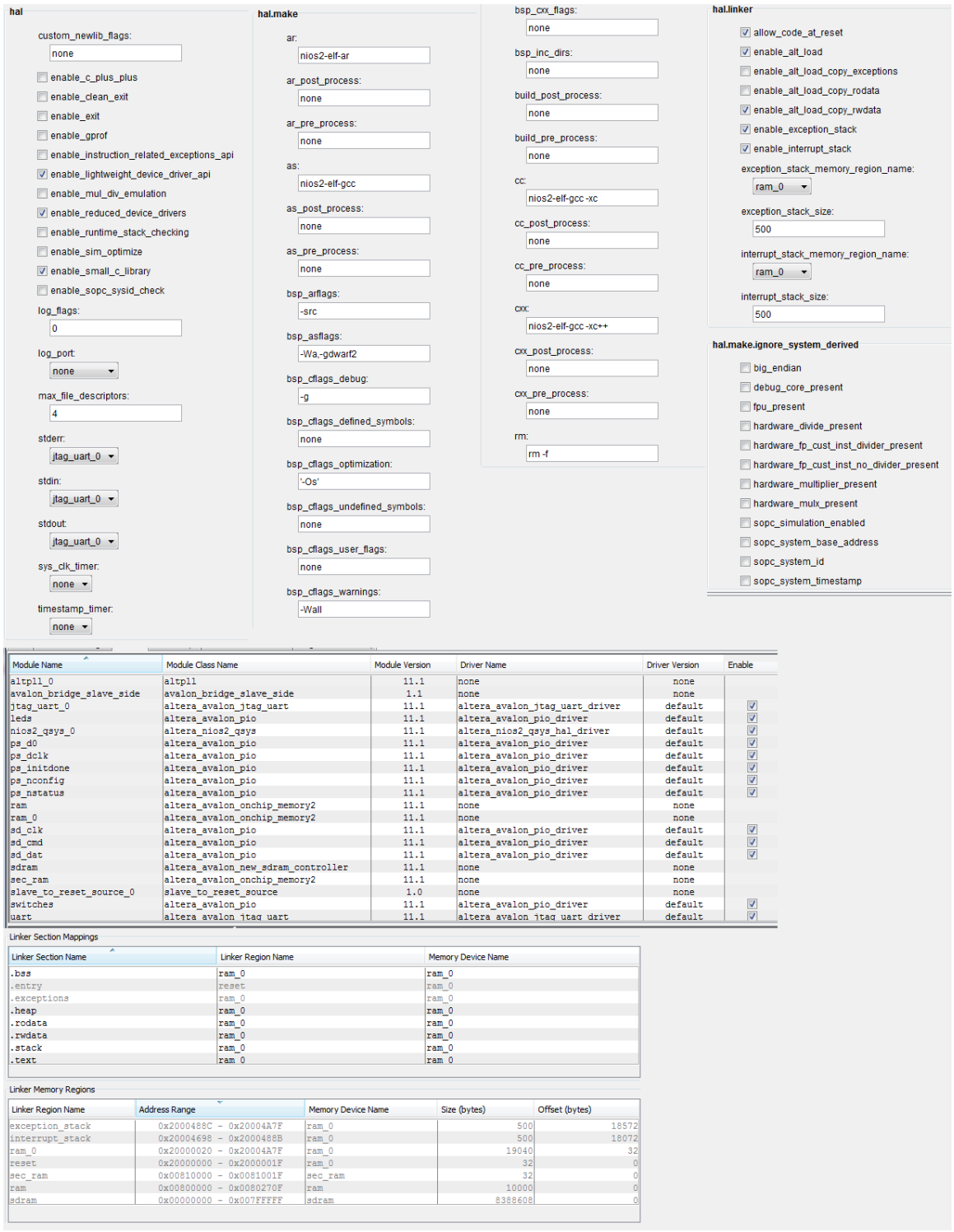

Figure 13. Security Controller BSP

All unnecessary functions are omitted, such as support for C++. Altera offers a set of drivers with reduced capability that have a smaller code size, and these are selected. No special exit from the code is needed as this application runs in a continuous loop. Inputs and outputs are directed to jtag_uart_0 such that they can be viewed from the PC. Code optimization is disabled; in this multiprocessor situation, the complier may incorrectly assume a code block has no effect, when actually the second processor is accessing the same memory locations for information. A small interrupt stack and a small exception stack are included. Interrupts are not used in this design to simply the implementation.

Figure 13 also shows the linker settings. This specifies where the actual hex code for the Nios II resides. Ram_0 is the Security Controller RAM and the entire application is located in this region. Note the eight linker sections: .bss, .entry, .heap, .rodata, .rwdata, .stack, and .text. Each of these can be individually placed in any memory accessible to the Nios II, with the exception of .entry and .exceptions, which are defined when instantiating the Nios II in Qsys. The size and address range of each of the sections is shown as well. Note that sec_ram is a 32-byte shared RAM that both the user Nios II and the Security Controller Nios II access.

Once defined, the BSP is compiled and the main software application references it.

5.1.2 Software Overview

The Security Controller software design is shown in flow-chart form in Figure 14a through Figure 14c. Upon reset, the Nios II starts at its .entry address as defined in the SoPC design. From there it jumps to .text and begins execution of the BSP functions, which finally allow the program to enter main( ) where it performs the process shown in Figure 14a through Figure 14c.

The first task is to ensure that the user application’s communication is inhibited. This is a double check as no user application should be allowed at this point as the user FPGA is not configured. Next, the security ratchet is checked to be zero. Upon a tamper event, the code will reenter as before, but the security ratchet will be higher than zero, which will cause the process to terminate. This termination is simply an infinite loop.

The ratchet is set to one, confirming that the Security Bridge is disabled and the user processor is halted. This progression of boot steps with incrementing ratchet values is an implementation of the trust ratchet as developed by the IBM 4758 team.

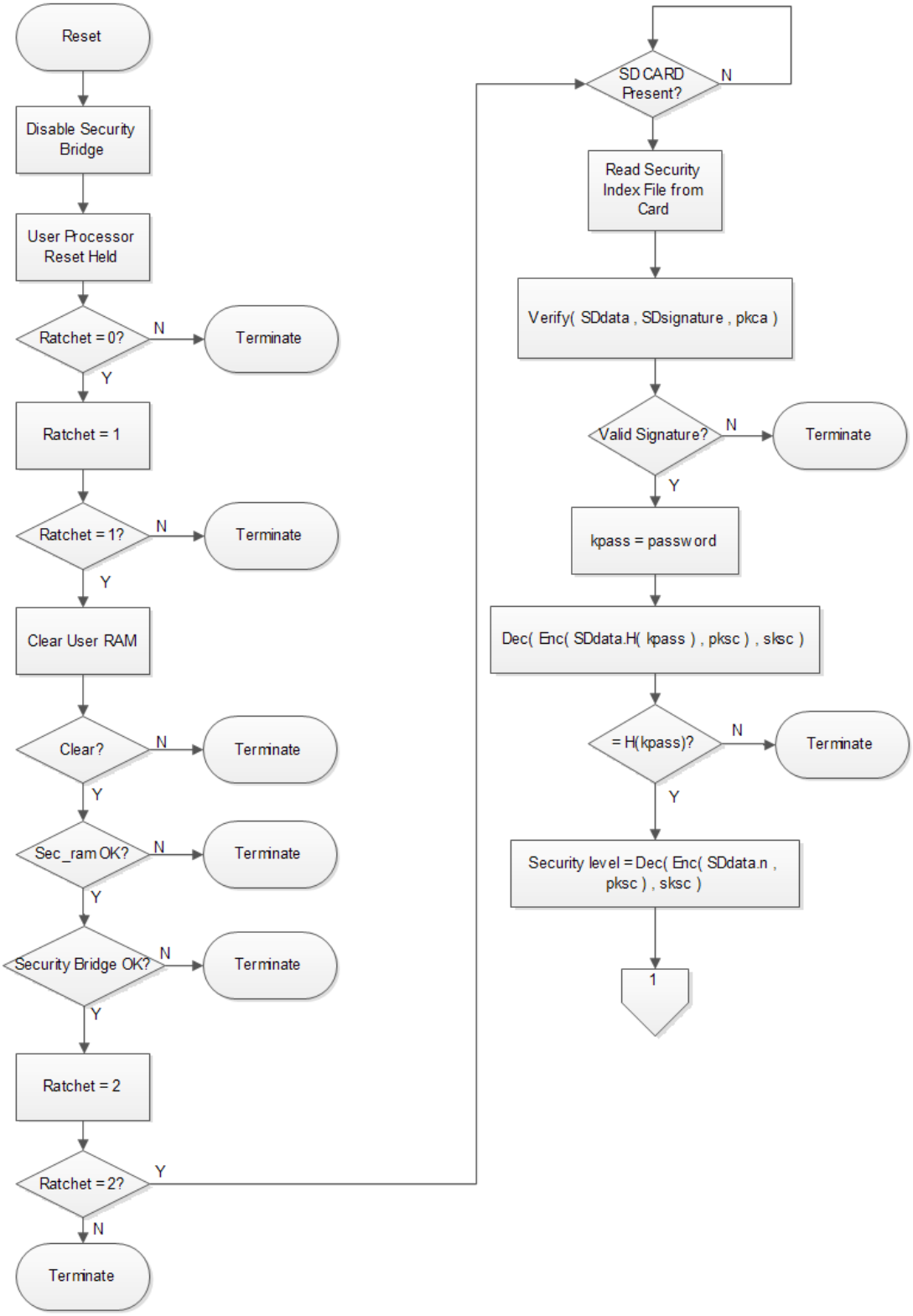

Figure 14a. Security Controller Software Flow

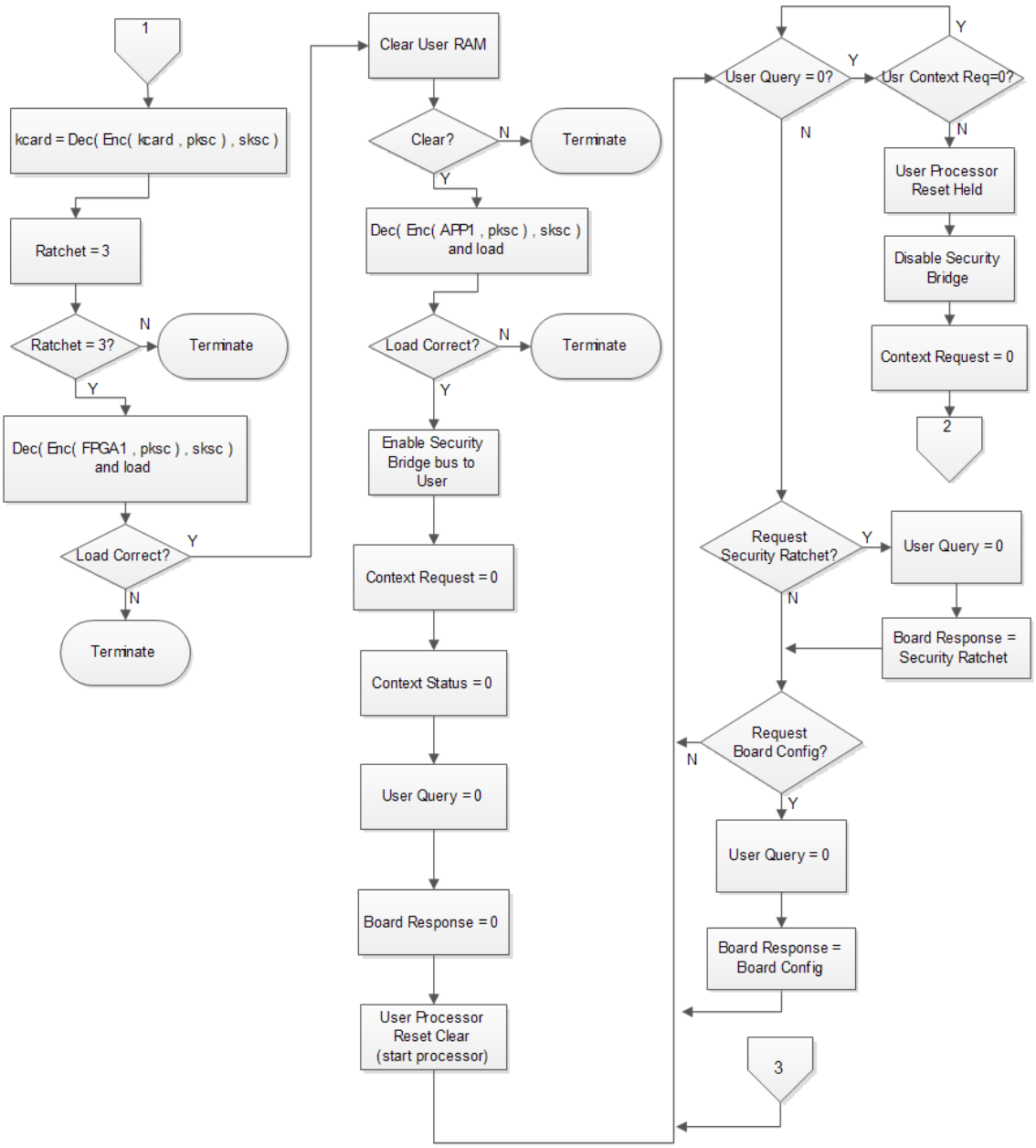

Figure 14b. Security Controller Software Flow

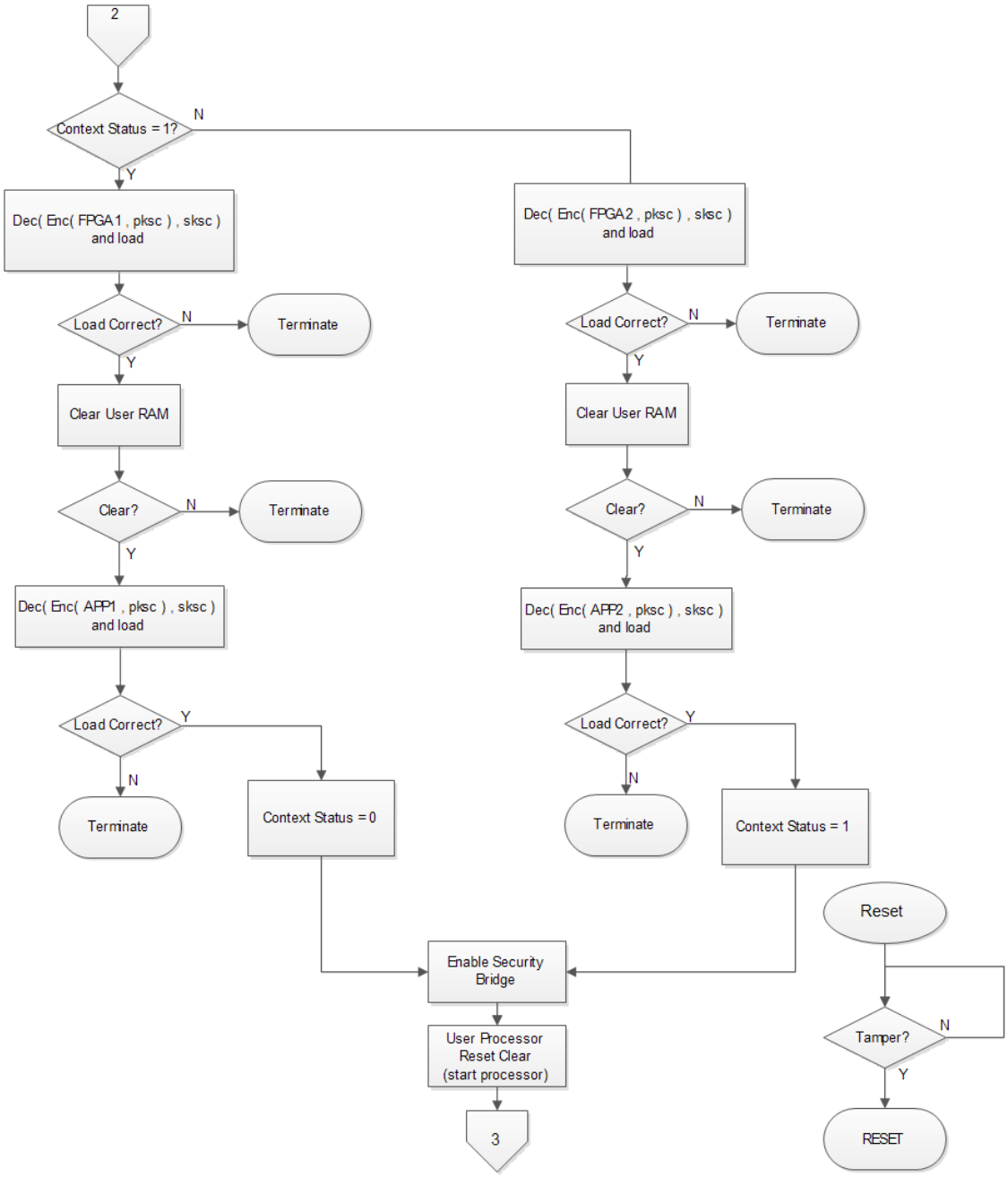

Figure 14c. Security Controller Software Flow

Next, the DE2 Main-Board hardware is checked. The user RAM area is cleared and checked and the 32-byte security RAM area is written to and read from to confirm its operation. The Security Bridge is checked by a similar process of write-read-verify.

If these operations succeed, then the security ratchet is set to two and the processor waits for the SD card to be inserted.

The SD card communication is accomplished by manipulating PIO lines directly from the processor. An SD library from [68] is incorporated, in part, to initialize and read the SD card. Additional functions are designed on top of this library to read and cache an entire 512-byte block from the file system, as well as various functions to convert ASCII to hex.

Next, the user enters k_{pass} at the terminal window. Then a security index file is read (this is depicted in Figure 5). This file contains file pointers to the FPGA configurations, the user applications, the allowed security level file, the k_{card} file, and the hashed password file. It also has the signature of the system owners attesting to the validity of those encrypted files, which are encrypted with pk_{sc}.

securityheaderexample
SLVL 53d5cf41
SGK1 53d5cf00
SGK2 53D5CF81
CAFS 53d5cfc2
CAFE 53d73f87
CATS 53d8aba5
CATE 53d8d10a
CBFS 53d73fc2
CBFE 53d8ab59
CBTS 53d8d1c5
CBTE 53d8f72a
SID1 01cf1d9b
SID2 00000000
SID3 00000000
SID4 00000000
SID5 00000000
SID6 00000000
SID7 00000000
SID8 00000000
endofexample

Text at the start and end are comments and are ignored. Each entry has a four-letter mnemonic followed by eight hex values. *SLVL* is the security level file pointer. *SGK1* is a pointer to the file containing k_{card}. *SGK2* is a pointer to the file containing $H(k_{pass})$. *CAFS* and *CAFE* are pointers to the FPGA1 file endpoints. *CATS* and *CATE* are pointers to the APP1hex file endpoints. *CBFS, CBFE, CBTS, CBTE* serve the same roles but for FPGA2 and APP2. *SID1* is the CA's signature

over the concatenation of all of the encrypted files. The index file is not encrypted as it contains the signature for the encrypt-then-sign protocol that is used for the data on the SD card, so this value must be separate from the encrypted data. The file pointers could be encrypted in their own file, but they are only indexes to files that are encrypted and signed. To change the pointer, one would have to forge the SD card signature and place that value in *SID1*, which is assumed hard.

Using these file pointers, $Verify(\ SDdata\ ,\ SID1\ ,\ pk_{ca}\ )$ is checked. The system halts if this fails. This is a two-party encrypt-then-sign scheme [19].

Then the user password is received, hashed, and checked against the decryption of the hash stored in the password hash file. This system halts if this fails.

Next, the security level is determined by decrypting the security level file and using that number. The access control policy is set. The usefulness of the custom hardware designed in Chapter 3 is now apparent as the access policy can easily be changed in real-time. It takes the form of a set of registers whose names are explanatory.

```
#define SEC_CONTROL_COM_ENABLE (unsigned int *) 0x10010000
#define SEC_CONTROL_RAMSEG1_LOW (unsigned int *) 0x10010004
#define SEC_CONTROL_RAMSEG1_HIGH (unsigned int *) 0x10010008
#define SEC_CONTROL_RAMSEG2_LOW (unsigned int *) 0x1001000C
#define SEC_CONTROL_RAMSEG2_HIGH (unsigned int *) 0x10010010
#define SEC_CONTROL_RAMSEG3_LOW (unsigned int *) 0x10010014
#define SEC_CONTROL_RAMSEG3_HIGH (unsigned int *) 0x10010018
#define SEC_CONTROL_ALLOW_UART (unsigned int *)  0x1001001C
#define SEC_CONTROL_ALLOW_SW (unsigned int *) 0x10010020
#define SEC_CONTROL_ALLOW_LEDS (unsigned int *) 0x10010024
```

These registers are filled with the access policy:

```
// Level 1
#define SEC_CONTROL_RAMSEG1_LOW1 0x200000
#define SEC_CONTROL_RAMSEG1_HIGH1 0x20025F
#define SEC_CONTROL_RAMSEG2_LOW1 0x204000
#define SEC_CONTROL_RAMSEG2_HIGH1 0x204007
#define SEC_CONTROL_RAMSEG3_LOW1 0x200263
#define SEC_CONTROL_RAMSEG3_HIGH1 0x200263

//Level 2
#define SEC_CONTROL_RAMSEG1_LOW2 0x200000
#define SEC_CONTROL_RAMSEG1_HIGH2 0x20025F
#define SEC_CONTROL_RAMSEG2_LOW2 0x204000
#define SEC_CONTROL_RAMSEG2_HIGH2 0x204007
#define SEC_CONTROL_RAMSEG3_LOW2 0x200264
#define SEC_CONTROL_RAMSEG3_HIGH2 0x200264
```

```
//Level 3
#define SEC_CONTROL_RAMSEG1_LOW3 0x200000
#define SEC_CONTROL_RAMSEG1_HIGH3 0x20025F
#define SEC_CONTROL_RAMSEG2_LOW3 0x204000
#define SEC_CONTROL_RAMSEG2_HIGH3 0x204007
#define SEC_CONTROL_RAMSEG3_LOW3 0x200265
#define SEC_CONTROL_RAMSEG3_HIGH3 0x200265

*SEC_CONTROL_ALLOW_UART = 0;
*SEC_CONTROL_ALLOW_SW = 1;
*SEC_CONTROL_ALLOW_LEDS = 1;
```

Next, the card key, k_{card}, is decrypted and saved into secure RAM. Then the security ratchet is incremented and the first FPGA configuration is decrypted and loaded. Should the user FPGA indicate a fault with the .rbf file load process, the Security Controller program will halt. The load function is designed to implement the Altera passive serial protocol [41].

Next, the application .hex file for the user Nios II is decrypted and loaded into a cleared user RAM. Then the Security Bridge is enabled, shared request and status registers are cleared and the user Nios II is allowed to start execution.

From this point forward, the Security Controller monitors the shared RAM for requests from the user Nios II. It can respond with a checksum of the board hardware to assure the application running on the user Nios II of the correctness of the run environment. It can also report the security ratchet to the User Nios II. It also responds to context change requests from the user.

When these are requested, the previous steps are mostly repeated, with a different FPGA .rbf file and application .hex file loaded, however. Figure 14a through Figure 14c show how this is handled.

5.1.3 Important Functions

This section briefly details the more important software functions:

Decryptsp: This function simulates a public-private key system decryption. It takes three parameters, all 32-bit unsigned represented. The first is the value to decrypt. The second combined with the third are the private key.

$$M = Decryptsp(M_{enc}, sk_1, sk_2), \text{ where } sk = sk_1 \,||\, sk_2$$

The cryptosystem is represented by a major simplification:

$$K_1 = sk \cdot pk$$
Where K_1 is a fixed value for the thesis
$< sk , pk >$ is a public-private key pair and are integers

The decryption process finds:

$$M = M_{enc} \oplus (sk)$$

This data is encrypted previously by the function:

$$M_{enc} = M \oplus (\frac{K_1}{pk})$$

In this manner, public-private key cryptography can be represented as a template function with unique public and private keys and the exclusive-or function.

GetCardKey : This function decrypts the card key from the SD card using the board public key:

$$k_{card}[31:0] = Decryptsp(Enc(k_{card}[31:0] , pk_{sc}) , sk_{sc})$$
$$k_{card}[63:32] = Decryptsp(Enc(k_{card}[63:32] , pk_{sc}) , sk_{sc})$$

DetermineSecurityLevel: This function gets the security level by decrypting the security level file:

$$n = Decryptsp(Enc(n , pk_{sc}) , sk_{sc})$$

LoadApp: This function takes start and stop file pointers to the SD card and downloads and decrypts the user application file using the board key:

$$instruction[31:0] = Decryptsp(Enc(instruction[31 : 0] , pk_{sc}) , sk_{sc})$$

Config: This function takes a file start and a file stop pointer to the SD card and decrypts the FPGA configuration file and loads it to the user FPGA and checks for problems. The file is decrypted using the board key exactly as in LoadApp, with an exception being that it is done byte-wise instead of word-wise.

CRC_SD and checksum: These functions simply add up data into a 32-bit unsigned register; this is a simplification of a cryptographic hash.

VerifySD: This function takes a message, a signature, and a public key and performs signature checking:

$$valid = VerifySD(signature , pk)$$

The simplification behind this is verifying:

$$signature \stackrel{?}{=} checksum(\ SDdata\ ) \oplus (\ \frac{K_2}{pk}\ )$$

Creating the signature take form:

$$signature = checksum(\ message\ ) \oplus (\ sk\ )$$

$$Where\ K_2\ is\ a\ fixed\ value\ for\ the\ thesis$$
$$K_2 = sk \cdot pk$$
$$< sk\ ,\ pk > is\ a\ public\text{-}private\ key\ pair\ and\ are\ integers$$

In this manner, signing and verifying a signing operation are simulated.

5.2 User Applications

The user software is designed to display the features of the entire platform. The Qsys SoPC system, when used with the Eclipse IDE allows the actual code to be fully simulated in ModelSim once the environment is carefully set up. The user applications are designed to both run the hardware and to run in the simulator, and the selection is made by compiler directives.

5.2.1 Board Support Package

As seen in Section 5.1.1, it is necessary for the design of the user application to be built upon a BSP designed for the goals in mind. Figure 15 shows the BSP for the user applications. It varies from the Security Controller BSP as its code segments are divided across two devices: ram_1, which is the tightly coupled RAM attached directly to the user Nios II, and userram, which is an alias for the user RAM whose access is mediated by the Security Bridge access control policy.

The standard input / output device is set to jtag_uart1. This communication port is directly attached to the user Nios II for debug purposes and would not be desired in a full implementation of this system due to security concerns.

All linker regions except .text, which is the main user application, are located in ram_1 that is attached to the Nios II directly. The user application .text segment is located in the user RAM area and is loaded by the Security Controller. Ram_1 is loaded during FPGA configuration.

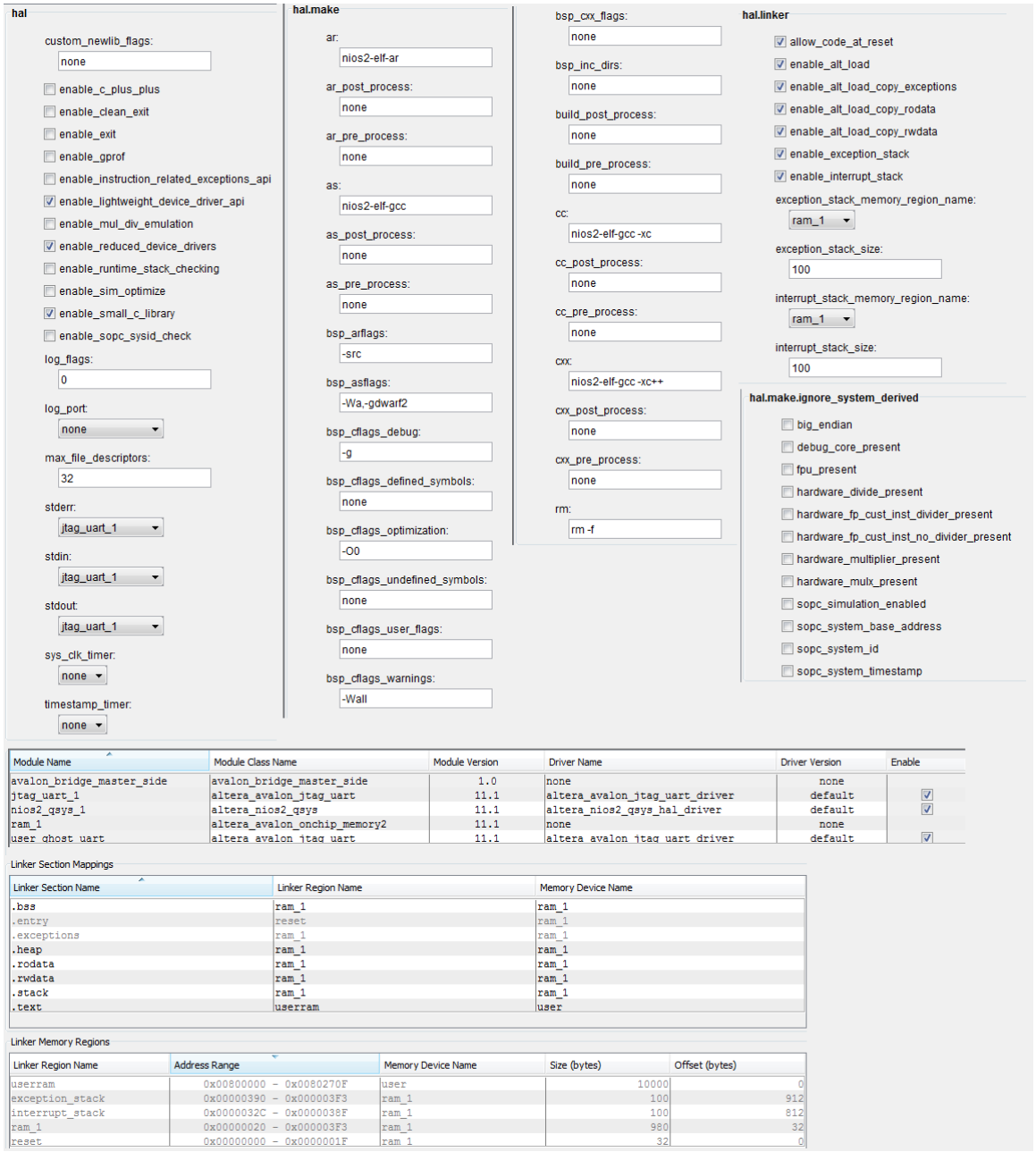

Figure 15. User Application BSP

5.2.2 Software

Refer to Figure 16. Upon reset, code executes from ram_1 then jumps to user RAM through the Security Controller. The Nios II then requests the current value of the security ratchet and checks that it is correct. It then requests a checksum of the DE2 Main Board and checks that value before continuing. Then it copies the pass key and the card key to the User Bridge. Next, it reads and writes to three test locations. The first is allowed and the remaining two are disallowed. Also, the actual code to perform the first test is encrypted, so the User Bridge decrypts and executes it. Next, the LEDs are set based on the user application—for application 1 they are set to one value and a different value for application 2.

Finally, the application requests that the Security Controller load the next configuration. The application then halts. Application 2 is identical except for the value of the LEDs displayed.

5.3 Encryption Software

The Quartus II toolset output files and the Eclipse output files must be post-processed for use with this system design. Custom programs are designed in Scilab to accomplish this. This post-processing approximates the processing needed as described in Section 5.1.3.

Code processing: This program asks the user which code segment has secondary encryption. Then it takes the user application object file and strips out all except the addresses and the instruction codes.

Next, it parses the file. If a particular segment is inside the secondary encryption segment, then it is encrypted as

$$inst_{enc}[31:0] = Inst[31:0] \oplus (k_{pass}[47:44] \,||\, k_{card}[47:44] \,||\, k_{pass}[43:40] \,||\, k_{card}[43:40] \,||\, k_{pass}[7:4] \,||\, k_{card}[7:4] \,||\, k_{pass}[3:0] \,||\, k_{card}[3:0])$$

Next, all code is encrypted as:

$$inst_{enc}' = inst[31:0] \oplus (\frac{K_1}{pk_{sc}})$$

FPGA configuration processing: this program takes a user FPGA.rbf file and encrypts it:

$$FPGAbyte_{enc}[7:0] = FPGAbyte \oplus (\frac{K_1}{pk_{sc}})$$

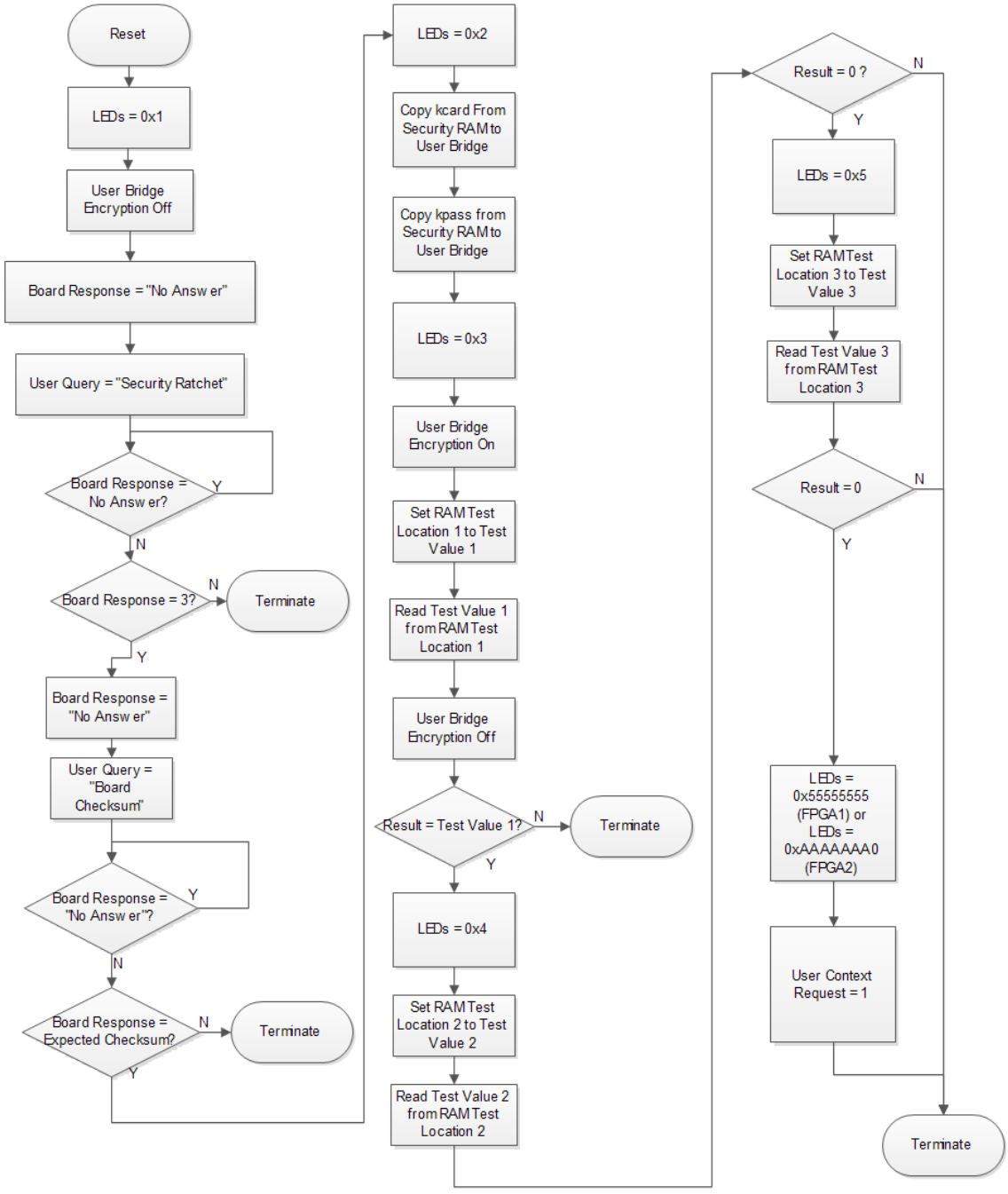

Figure 16. User Application

Data Signing: to model the CA signature of the encrypted data, SID1 on the SD card is set to:

$$SID1 = checksum(\ SDdata) \oplus (\ sk_{ca}\ )$$
$$where\ SDdata\ is\ FPGA1\ ||\ FPGA2\ ||\ APP1\ ||\ APP2\ ||\ security\ level\ assigned\ ||\ checksum(k_{pass})\ ||\ k_{card}$$

5.4 Ancillary Software

Multiple small applications are designed to run the entire development platform to automate the tasks of FPGA synthesis, placement, routing, loading, rbf file generation, hex file generation, application of the encryption programs, SD card loading, and correct placement of files for ModelSim.

CHAPTER 6
RESULTS

The complete system as described is run successfully on hardware and in ModelSim. This chapter details the results from the tests with the hardware system as well as simulation results that show the internal operation clearly.

The simulation results are used show the totality of the system operation as well as interesting portions of important modules as they interact with the system. Some aspects of the simulation are not modeled to reduce the model complexity, such as modeling of the SD card. Other aspects cannot be modeled, such as FPGA reconfiguration. In cases where simulation simplifications are made, testing relies on actual hardware to show the results. Simulation provides results that are not seen by observation of the hardware, such as zeroing of secrets. Taken together, they show the full system operation.

6.1 Full System Operation

First, the full system is generated. The tasks to coordinate a system built are numerous and use multiple tools.

1. A full set of keys are generated by manual selection of hex digits.
2. The user FPGA1 configuration is generated in a special, separate project environment.
3. The user FPGA1 file is run through Scilab to encrypt it with the board key.
4. The user FPGA1 file is altered to output a different set of LEDs on the DE2 Sub-Board and is saved as FPGA2.
5. The user FPGA2 file is run through Scilab to encrypt it with the board key.
6. The FPGA files have markers added to them so that they can be located on the SD card.
7. A checksum of the password key is made and placed in a file. The file is encrypted with the board key. A marker is added to the file to locate it.
8. The user's security level is looked up and placed in a file. The file is encrypted with the board key. A marker is added to the file to locate it.
9. The user's key card key is placed in a file. The file is encrypted using the board key. A marker is added to the file to locate it.
10. The FPGA1 file, FPG2 file, key card file, password hash file, and security level file are placed on the SD card and located manually using the markers. These locations are noted for placement in the security index file.
11. The Security Controller Nios II application is compiled and the hex code is generated.
12. The full Security Controller FPGA is compiled in Quartus using a custom script to synthesize, map, place, and fit the SoPC. It also takes the hex

code from step 11 and inserts it into the FPGA configuration so the Security Controller Nios II starts correctly when the FPGA is programmed.
13. A checksum of the Security Controller ROM is calculated.
14. The checksum is inserted in the user APP1 and APP2 program as the expected board checksum value.
15. The user APP1 program is selected by a pre-compiler directive. It is compiled, and the hex code generated.
16. A Scilab program is executed that that takes APP1 and asks the user which code addresses to apply secondary encryption to, such that when the application decrypted and loaded by the Security Controller, that segment is still encrypted. That program then encrypts that code segment with the pass key and the card key.
17. Next, the Scilab program encrypts the entire program with the board key.
18. Steps 14-17 are repeated for the user APP2 program.
19. The Scilab program adds markers to APP1 and APP2 to locate them on the SD card and it loads them onto the SD card.
20. A security index file is generated. To do this, the files loaded on the SD card have to be located. This is accomplished with a very useful program called HxD that allows direct access to drive data. Pointers to the files are added to the security header file.
21. A signature of the encrypted files is made using the private key of the CA sk_{ca}. This is added to the security index file.
22. The security index file is inserted in a specific place on the SD card using HxD.
23. The system is powered off and on and the SD card is inserted.
24. The system is programmed from Quartus. This programs the Security Controller FPGA.
25. A terminal window is opened in the Eclipse IDE. A connection is made to jtag_uart0.
26. System operation begins and is recorded as follows.

Figure 17 shows the complete system prior to start up. A laptop is connected via USB to the device port on the DE2 Sub-Board. The system is booted and the password, k_{pass}, is entered; “x” demarks the start of the password:

x000f000f000f000f

Then the system runs the described system checks. No further user intervention is needed to demonstrate the system.

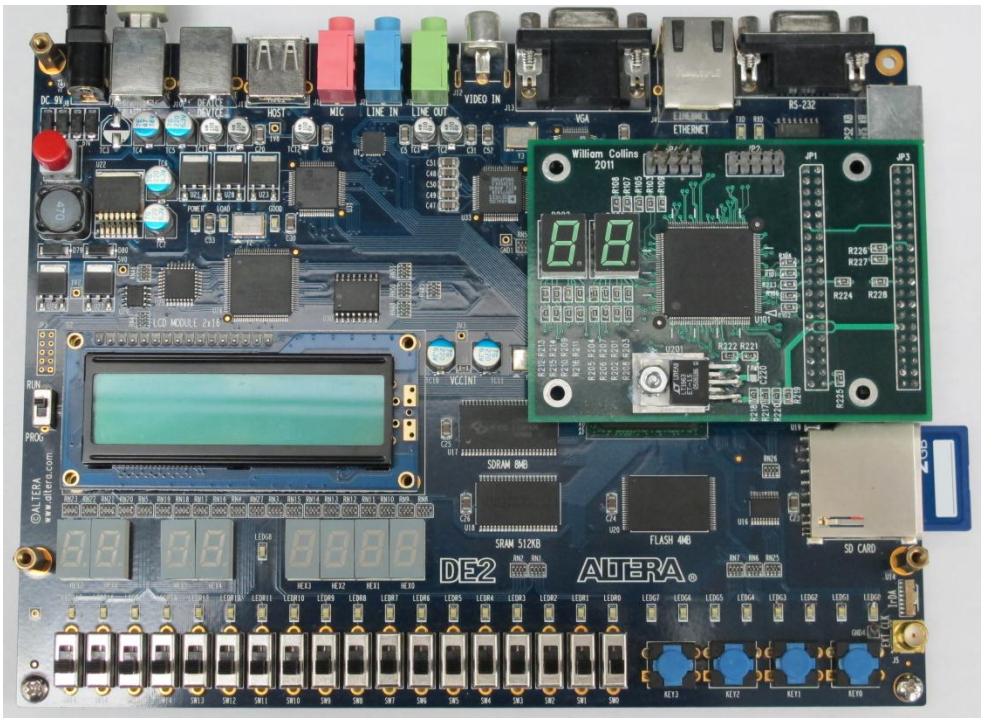

Figure 17. The Full System

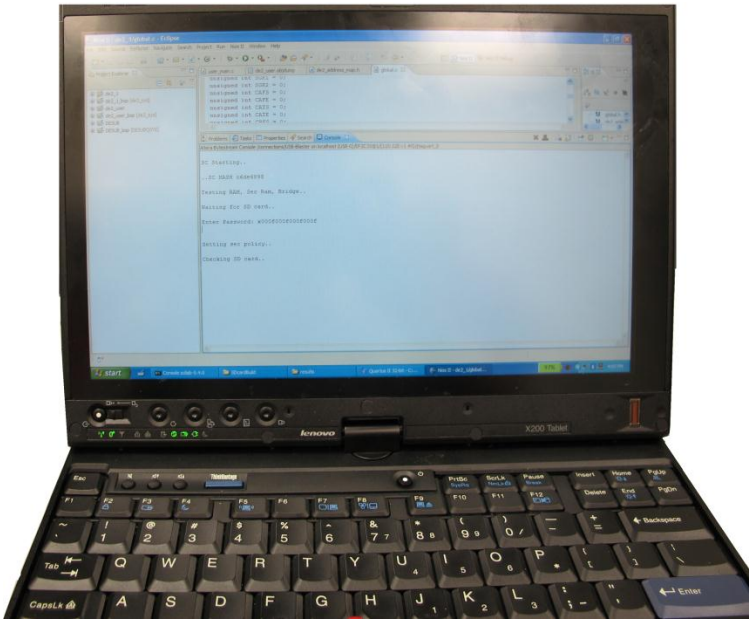

Figure 17. Continued.

The terminal informs the user of the status:

SC Starting..
..SC HASH 9bc6b732
Testing RAM, Sec Ram, Bridge..
Waiting for SD card..
Checking SD card..
..SD signature 1cf1d9b
..SD calculated signature 1cf1d9b
Enter Password: x000f000f000f000f
Setting sec policy..
Loading FPGA A..
Loading APP A..
Starting Processor A
Sec Ratchet 3 sent to USER
Board Hash 9bc6b732 sent to USER
User Requested Context Change
Stopping User Processor..
Loading FPGA B..
Loading APP B..
Starting User Processor..
Sec Ratchet 3 sent to USER
Board Hash 9bc6b732 sent to USER
User Requested Context Change
Stopping User Processor..
Loading FPGA A..
Loading APP A..
Starting User Processor..
Sec Ratchet 3 sent to USER
Board Hash 9bc6b732 sent to USER
User Requested Context Change
Stopping User Processor..
Loading FPGA B..
Loading APP B..
Starting User Processor..
Sec Ratchet 3 sent to USER
Board Hash 9bc6b732 sent to USER
User Requested Context Change
Stopping User Processor..
Loading FPGA A..
Loading APP A..
Starting User Processor..
Sec Ratchet 3 sent to USER
Board Hash 9bc6b732 sent to USER
User Requested Context Change
Stopping User Processor..
Loading FPGA B..
Loading APP B..
Starting User Processor..
Sec Ratchet 3 sent to USER
Board Hash 9bc6b732 sent to USER
User Requested Context Change
Stopping User Processor..

“SC Starting” indicates the reset of the Security Controller Nios II. It then performs a checksum on its program area and outputs the value. This value is later checked by the user software. Next, the Security Controller performs system checks. It clears the user RAM area and checks that it is clear. It checks that the security RAM can be written to and read. It checks that the Security Bridge registers work. It then requests an initialization from the SD card. After that, it calculates the signature of the SD card data based on the system owner’s public key. It checks this value against the one published on the SD card and halts if there is a mismatch. It then waits for the user to enter the password. Following that, it determines the user’s authorization level (1, 2, or 3) and sets the access policy accordingly. Then it loads the first FPGA configuration. It loads the user application 1 and allows the user processor to start. The user processor starts and immediately requests the value of the security ratchet and checks it. It then requests a checksum of the state of the board and checks this against the expected value in its code. It then attempts three writes (not shown) to three different addresses, which are at each security level (1, 2, and 3), and verifies that it can write only to its level. It then requests FPGA configuration 2 and application 2. The security controller loads them and restarts the processor. This process repeats continuously to demonstrate the system.

The complete flow from Figure 14a through Figure 14c and Figure 16 can be followed from this output.

Figure 18 shows the state of the board after the FPGA has been configured for configuration 1 and application 1 is running. A green LED on the DE2 Sub-Board is illuminated by the user FPGA as it has been configured and is operational. The green LEDs on the bottom indicate *55h*, which is commanded by the user application 1.

Figure 19 shows the state of the board again, except after it has been configured with the second FPGA configuration and the second user application. Now the seven-segment LED on the DE2 Sub-Board is fully illuminated as this is the output from FPGA configuration 2, and the LEDs on the bottom of the board display *A0h* as commanded by the user Nios II second application.

Pushbutton 0 on the DE2 Sub-Board is pressed at various times to verify the operation of the system under simulated tamper events. Repeated restarts still cause the system to halt. It takes a reprogramming to clear the security ratchet:

Checking SD card..
SC Starting..

..SECURITY FAILURE: halting..
SC Starting..

..SECURITY FAILURE: halting..

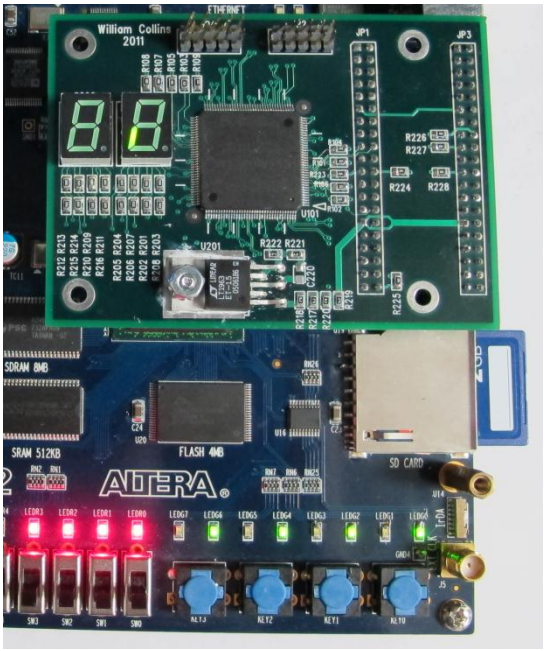

Figure 18. Configuration 1

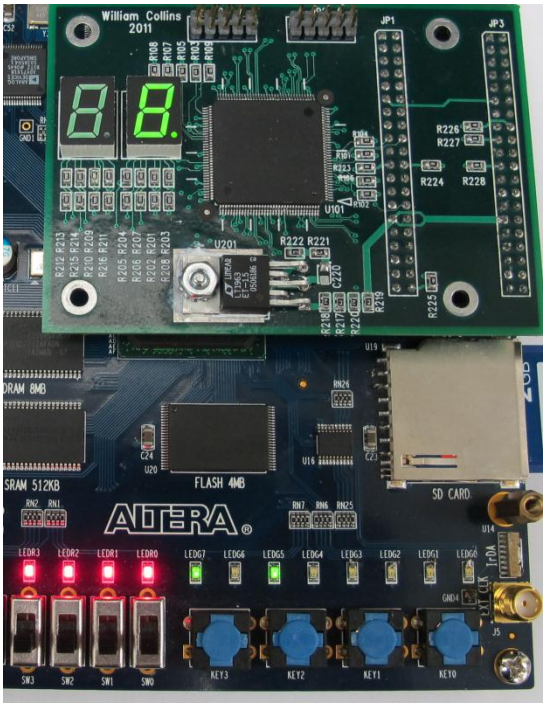

Figure 19. Configuration 2

Figure 20 shows the state of the DE2 Main Board after the simulated tamper event. The green LEDs indicate *77h* and this is a code from the Security Controller when it is in a halt state.

The full system operation is simulated in ModelSim with minor changes. The SD card is not simulated. Instead, information that it provides to the Security Controller is inserted into the code directly by direct variable assignments. This occurs when the simulation pre-compiler directive is defined. The user password entry is forced to make the system simulate faster. Additionally, the switches are forced to zero. The display of messages from the UART is inhibited to reduce the number of cycles needed to simulate. The FPGA configuration is not simulated, and as an extension of that, only one user application is simulated, but it provides adequate opportunity to observe the operation of the system internals.

Figures 21a through 21c show the full operation of the system through a user Nios II reconfigure command and a forced security fault. The organization is from top to bottom: the Security Controller Nios II and its components start the figure, then below is the User Nios II and its required components, and below that is the User Bridge. Then the Security Bridges follows and connects to the user Ram and Security RAM.

Starting at the top, the Security Controller Nios II instruction bus is accessed from power up until the security fault occurs, at which point it resets. It is getting instructions from ram_0, its nearby RAM. The Security Controller Nios II data bus is accessing that RAM as well to access the stack and variables. Also, the data bus is accessed to communicate with all of the security objects to initialize them.

Skipping past the local RAM, the User Nios II Reset Control is seen to command resets as directed by the Security Controller. The brief logic 1 times are when the User Nios II is commanded to reset. This occurs several times as the simulation is allowed to run through several configuration cycles.

By looking at the user Nios II instruction bus, it can be seen where the user Nios II comes out of reset and starts execution as commanded by the Security Controller. Its instructions are coming from the user RAM through the User Bridge and the Security Bridge. The user Nios II data bus also communicates with the User Bridge to interact with the security objects through the Security Bridge.

The User Bridge together with the Security Bridge serialize, transport, and deserialize the user Nios II’s instruction and data bus and decide if the access is allowed. The user RAM provides the application code to the user Nios II.

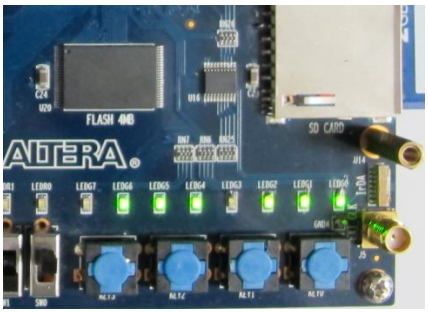

Figure 20. Security Fault State

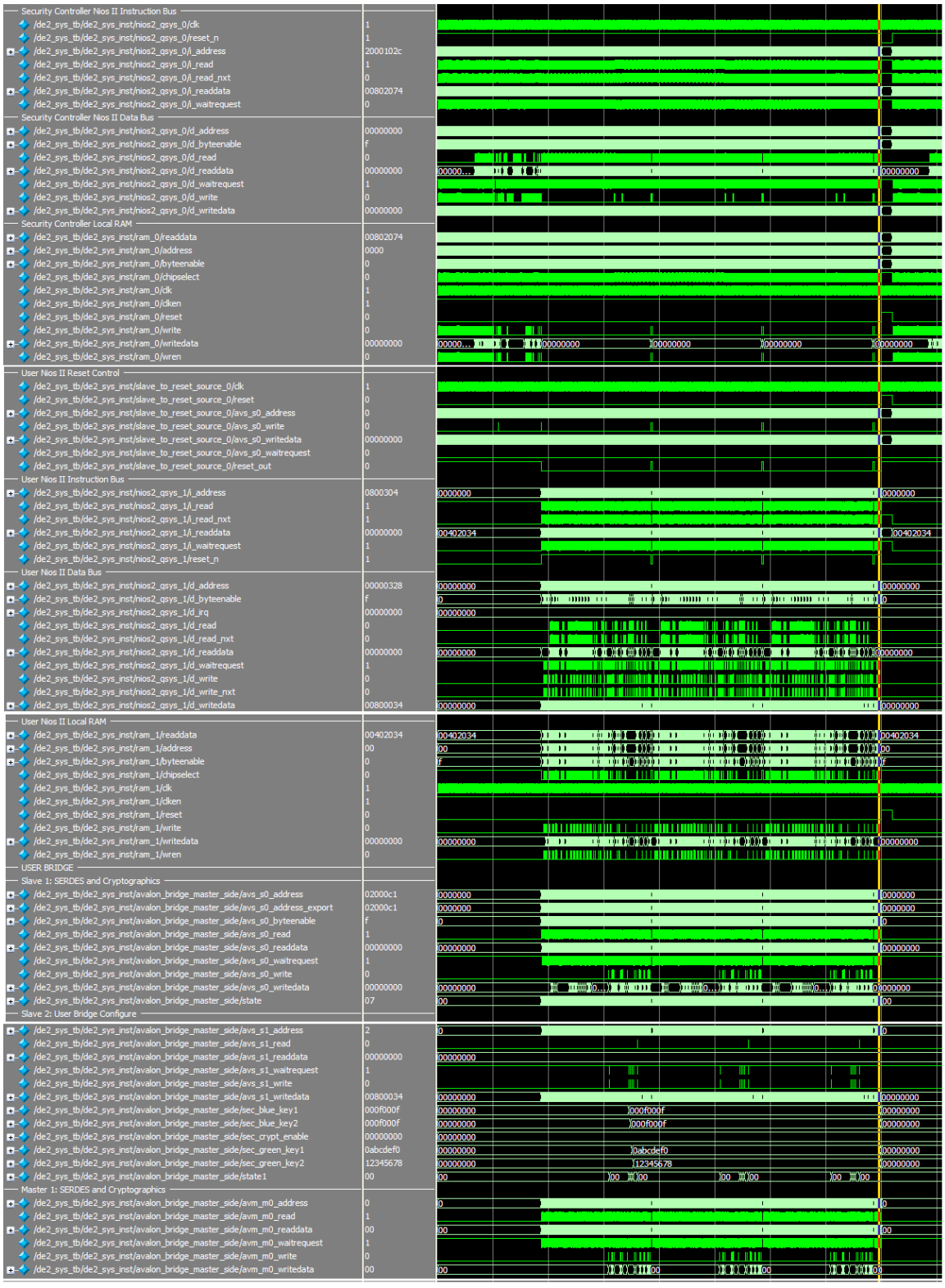

Figure 21a. Full System Simulation with ModelSim

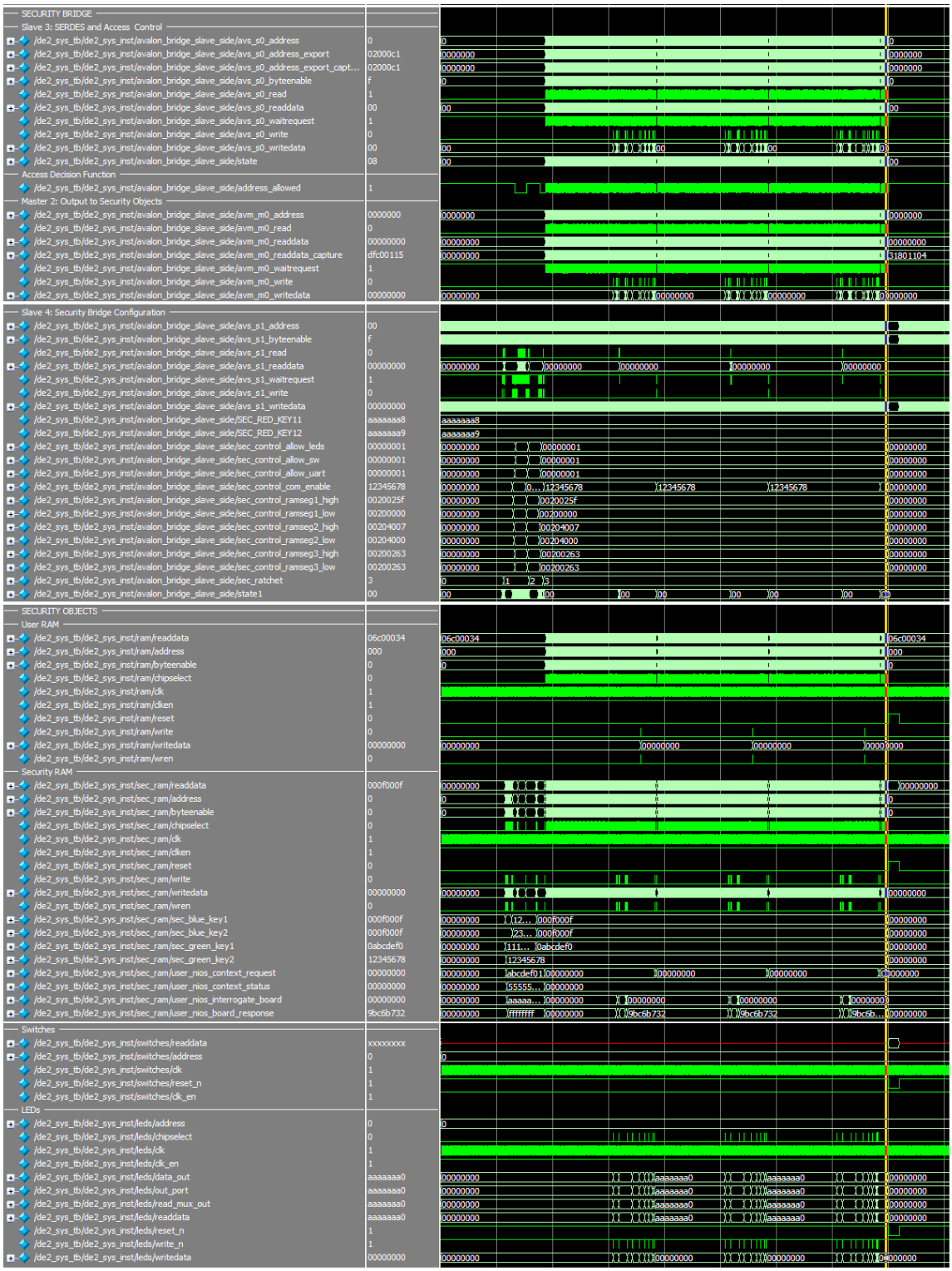

Figure 21b. Full System Simulation with ModelSim

Finally the security RAM, the User Bridge, and the Security Bridge secrets clear on the reset. This can be seen at in Figure 21b where the stored data changes to zeros and stay at zero.

6.2 Component Operation

This section details the cycle-level operation of the unique, key components of the system.

6.2.1 User Bridge

The User Bridge simulations show the configuration of the bridge, the serialization of the data crossing the bridge, and the encryption and decryption of the data to and from the user Nios II.

Consider the execution of this segment of code that copies the pass key (alias: blue key) from the security RAM to the User Bridge:

```
*SEC_BLUE_KEY1 = *USER_BLUE_KEY1;
 800120:    00c40074    movhi   r3,4097
 800124:    18e42804    addi    r3,r3,-28512
 800128:    00802074    movhi   r2,129
 80012c:    10800017    ldw     r2,0(r2)
 800130:    18800015    stw     r2,0(r3)
```

Pointers are used extensively so that the keys are only referenced from the designated locations that are cleared on reset. For the user Nios II to read the first instruction at *800120h*, it places this address on the instruction bus address and asserts a read. Figure 22 shows this operation with all extraneous detail removed. The Avalon fabric converts the global address, which is in bytes, to a word address for the User Bridge slave, so the address is converted to *200048h*. The User Bridge serializes this and outputs it on its own master port, which connects to the Security Bridge slave port (not shown.) The Security Bridge accesses the instruction from user RAM, serializes it, and places it back on the User Bridge's master bus, on the readdata bus. The User Bridge de-serializes the data and places it on its slave port readdata bus so the Avalon fabric can take it. The User Bridges takes waitrequest low to indicate the data is ready. The Avalon fabric transports the result to the Nios II instruction bus, where it appears under readdata. The Avalon fabric takes the wait request for this transaction low to tell the Nios II instruction that the data is ready. The cursor is placed at this point, indicating that a read of *00C40074h* is received by the Nios II, which is the first instruction in this sequence. Figure 23 shows a write operation across the User Bridge.

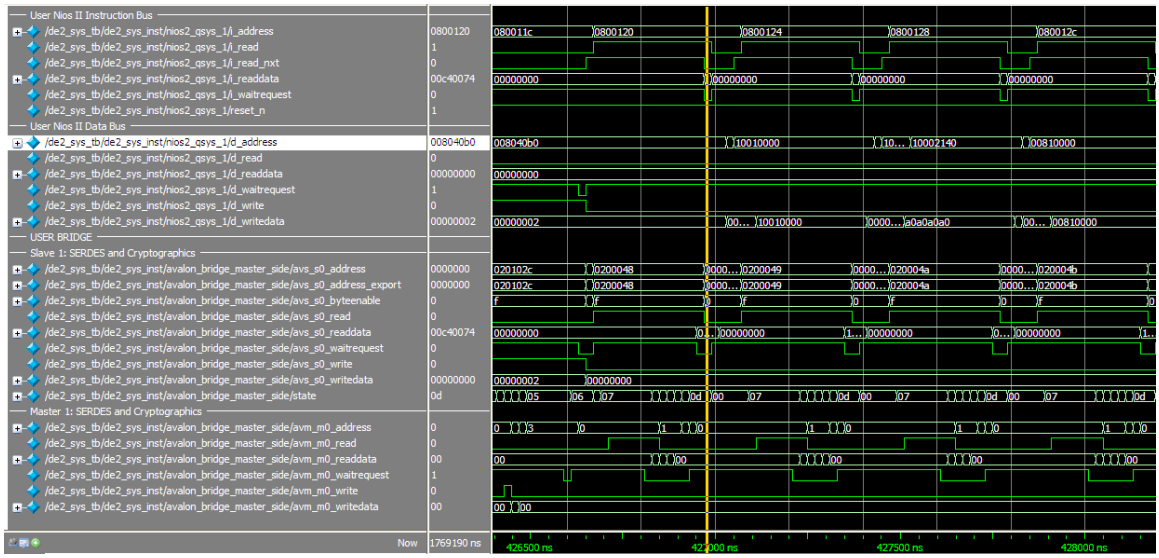

Figure 22. User Nios II Instruction Read without Encryption

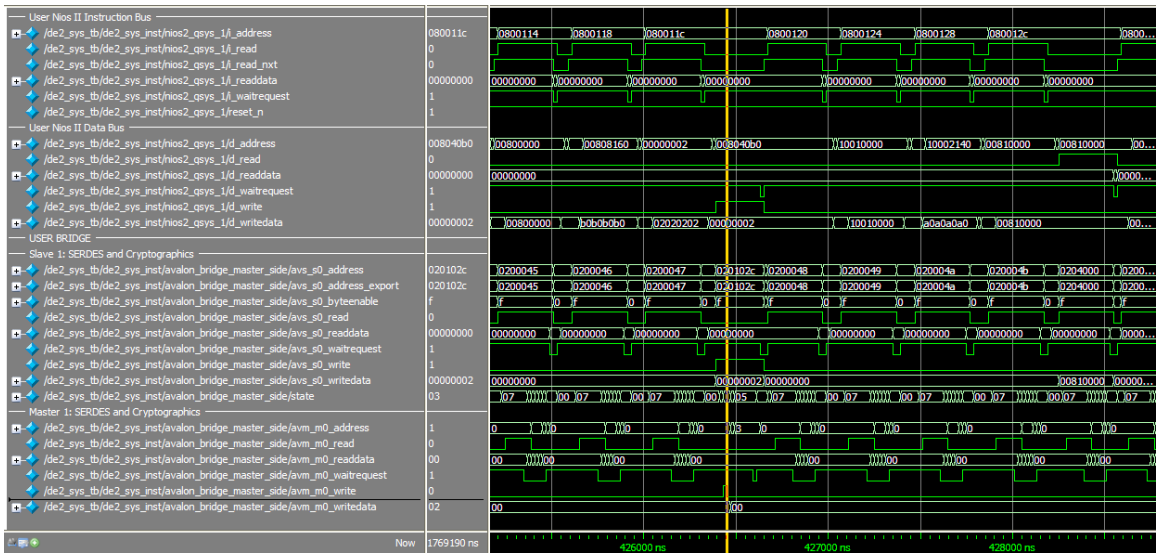

Figure 23. User Nios II Data Bus Write without Encryption

In this case, the LEDs are written to:

```
*LEDs = 0x2;
800110:     00c02034    movhi   r3,128
800114:     18d02c04    addi    r3,r3,16560
800118:     00800084    movi    r2,2
80011c:     18800015    stw     r2,0(r3)
```

The write takes place at instruction *80011ch* and the data value to be written is *2h*. This write originates from the user Nios II data bus instead of the instruction bus, and this *2h* is seen on the data bus with the address bus indicating *8040b0h*, which is where the LEDs are located. A similar process to the read operation occurs. The request is serialized and sent to the Security Bridge master port and the system does not continue until the wait request propagates from the Security Bridge, through the User Bridge, back to the user Nios II.

Figure 24 and Figure 25 show how data crossing the User Bridge is encrypted and decrypted according to Figure 5. The data is encrypted with a composite pass and card key. The difference between this operation and the operations in Figure 22 and Figure 23 is that the User Bridge has been commanded to encrypt the data. This allows the user application to selectively run encrypted code segments and to selectively encrypt data that is stored in RAM, without any additional processor interaction. During compilation, the following code segment is marked as encrypted and is encrypted by Scilab with the pass key and the card key:

```
        *RAMt0 = 0xace0ace0;
80018c:     00c02034    movhi   r3,128
800190:     18c26304    addi    r3,r3,2444
800194:     00ab3874    movhi   r2,44257
800198:     10ab3804    addi    r2,r2,-21280
80019c:     18800015    stw     r2,0(r3)
        a = *RAMt0;
8001a0:     00802034    movhi   r2,128
8001a4:     10826304    addi    r2,r2,2444
8001a8:     10800017    ldw     r2,0(r2)
8001ac:     e0bfff15    stw     r2,-4(fp)
        *SEC_CRYPTO_ENABLE = 0;
8001b0:     00840074    movhi   r2,4097
8001b4:     10a42c04    addi    r2,r2,-28496
8001b8:     10000015    stw     zero,0(r2)
```

RAMt0 is a location in user RAM that is allowed by the access policy. This code segment simply writes and reads a test word to *RAMt0* then disables the User Bridge Encryption. The code to enable the encryption is not shown, as it is not encrypted.

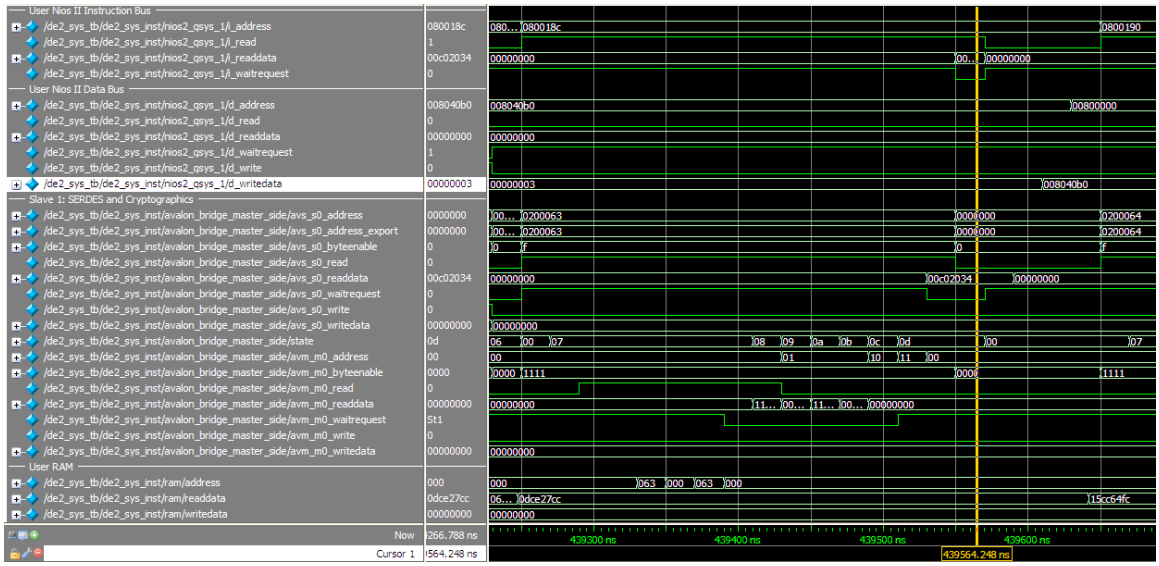

Figure 24. User Nios II Encrypted Instruction Read

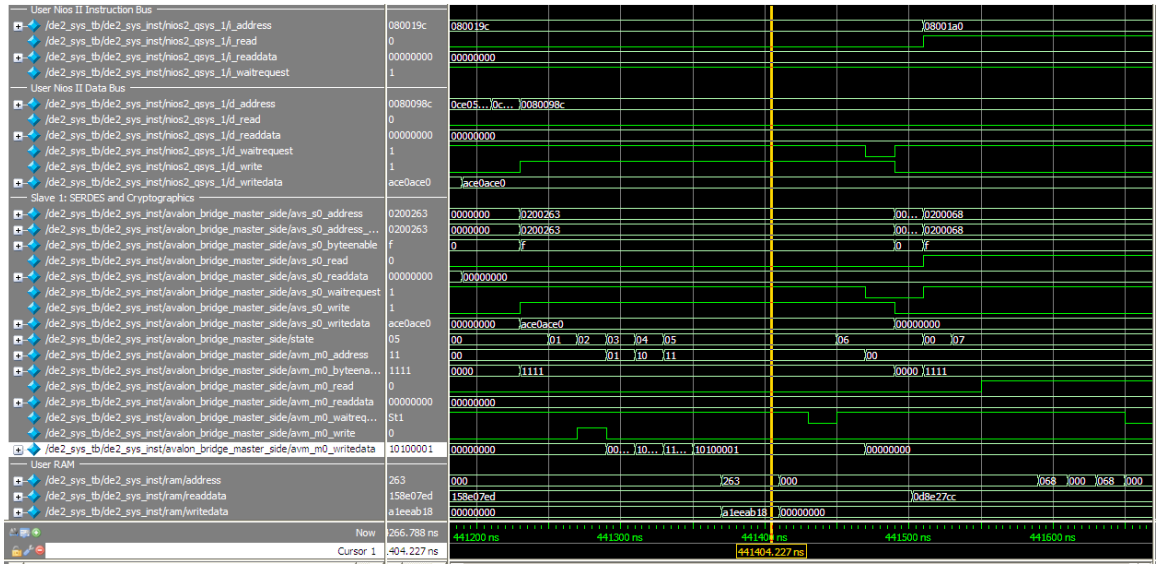

Figure 25. User Nios II Encrypted Data Write

The first encrypted instruction retrieved is *80018c: 00c02034 movhi r3,128*. Figure 24 shows *00c0234h* being received by the User Nios II instruction bus. The cursor is located on the reception of the wait request clear so that the returned value is displayed in the table. However, the user RAM value at *80018ch* is *0dec277ch.* The User Bridge has performed this transformation on the data as it crossed:

$$Dec(\ Inst\ ,\ k_{pass}[47:44]\ ,\ k_{card}[47:44]\ ,\ k_{pass}[43:40]\ ,\ k_{card}[43:40]\ ,\ k_{pass}[7:4]\ ,\ k_{card}[7:4]\ ,\ k_{pass}[3:0]\ ,\ k_{card}[3:0]\ ),\ where\ Inst\ is\ 00c02034h,$$
$$k_{pass}\ is\ 000f000f000f000fh,\ and\ k_{card}\ is\ 0ABCDEF012345678h.$$

An encrypted write is shown in Figure 25. Here *80019c: 18800015 stwr2,0 (r3)* is executed. The write is to *80098ch* with data *ace0ace0h*. This is encrypted to *a1eeab18h* as can be seen at the bottom of Figure 25. The User Bridge is configured by the user Nios II. Figure 26 shows the configuration in process. The pass key, upper and lower, has been filled in (shown as blue_key). The card key upper half is also filled in (shown as green_key). The current operation is writing the lower half of the card key, *0abcedf0h*.

6.2.2 Security Bridge

As with the User Bridge, the Security Bridge also has a slave port that allows its internals to be configured during run-time. In this case, the configuration is under control of the Security Controller Nios II instead of the user Nios II. Refer to Figure 27. The board keys are constants and can only be read. The next ten registers set what addresses are allowed to be accessed and their names are explanatory. Note the correlation with Table 5. The security ratchet can be seen incrementing from zero to three. Note that the registers in the Security Bridge are written to twice, with the final value being the correct configuration. The first set of write and reads is due to a software check on initialization that confirms that the registers are operational.

Next, consider this segment of code that illustrates the operation of the Security Bridge serialization, de-serialization, and access control functions.

```
*RAMt0 = 0xace0ace0;
 80018c:      00c02034      movhi   r3,128
 800190:      18c26304      addi    r3,r3,2444
 800194:      00ab3874      movhi   r2,44257
 800198:      10ab3804      addi    r2,r2,-21280
 80019c:      18800015      stw     r2,0(r3)
        A = *RAMt0;
 8001a0:      00802034      movhi   r2,128
 8001a4:      10826304      addi    r2,r2,2444
```

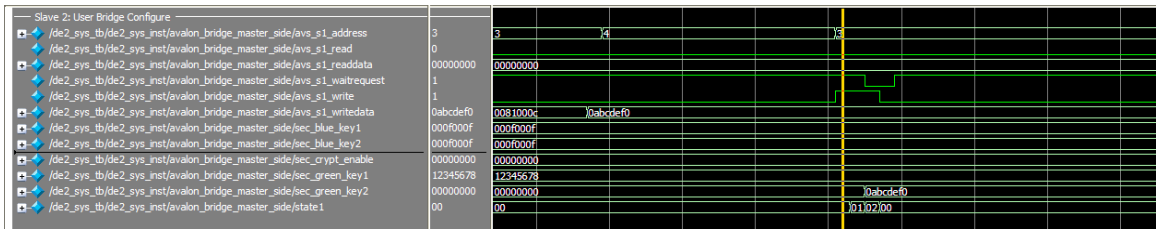

Figure 26: Configuring the User Bridge

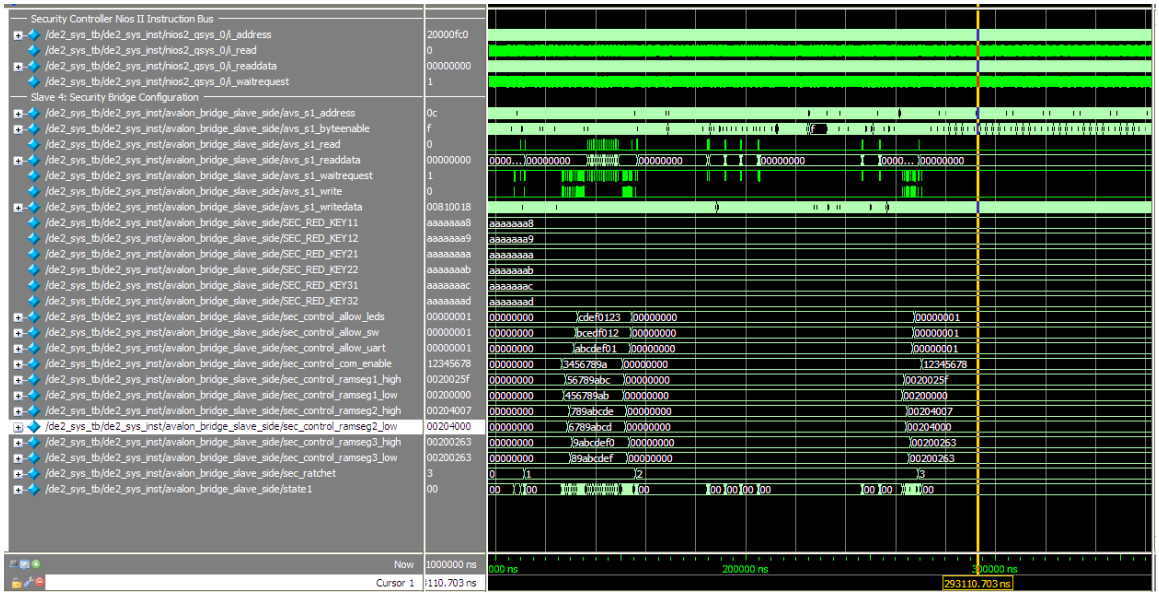

Figure 27: Configuring the Security Bridge

```
8001a8:   10800017   ldw   r2,0(r2)          See Figure 29.
8001ac:   e0bfff15   stw   r2,-4(fp)
        *SEC_CRYPTO_ENABLE = 0;
8001b0:   00840074   movhi r2,4097
8001b4:   10a42c04   addi  r2,r2,-28496
8001b8:   10000015   stw   zero,0(r2)
        if (a != 0xace0ace0)
8001bc:   e0ffff17   ldw   r3,-4(fp)
8001c0:   00ab3874   movhi r2,44257
8001c4:   10ab3804   addi  r2,r2,-21280
8001c8:   18800126   beq   r3,r2,8001d0 <main+0x114>
                halt();
8001cc:   080003c0   call  80003c <halt>
        *RAMt1 = 0xace1ace1;
8001e0:   00c02034   movhi r3,128
8001e4:   18c26404   addi  r3,r3,2448
8001e8:   00ab38b4   movhi r2,44258
8001ec:   10ab3844   addi  r2,r2,-21279
8001f0:   18800015   stw   r2,0(r3)          See Figure 30.
        a = *RAMt1;
8001f4:   00802034   movhi r2,128
8001f8:   10826404   addi  r2,r2,2448
8001fc:   10800017   ldw   r2,0(r2)          See Figure 31.
800200:   e0bfff15   stw   r2,-4(fp)
        if (a != 0x0)
800204:   e0bfff17   ldw   r2,-4(fp)
800208:   1005003a   cmpeq r2,r2,zero
80020c:   1000011e   bne   r2,zero,800214 <main+0x158>
                halt();
800210:   080003c0   call  80003c <halt>
        *RAMt2 = 0xace2ace2;
800224:   00c02034   movhi r3,128
800228:   18c26504   addi  r3,r3,2452
80022c:   00ab38f4   movhi r2,44259
800230:   10ab3884   addi  r2,r2,-21278
800234:   18800015   stw   r2,0(r3)          See Figure 32.
        a = *RAMt2;
800238:   00802034   movhi r2,128
80023c:   10826504   addi  r2,r2,2452
800240:   10800017   ldw   r2,0(r2)          See Figure 33.
800244:   e0bfff15   stw   r2,-4(fp)
        if (a != 0x0)
800248:   e0bfff17   ldw   r2,-4(fp)
80024c:   1005003a   cmpeq r2,r2,zero
800250:   1000011e   bne   r2,zero,800258 <main+0x19c>
                halt();
```

This user code attempts to write three different test words to three different addresses and it also attempts to read them back (Figure 28-33). The first address is allowed by the access control policy and the last two are not.

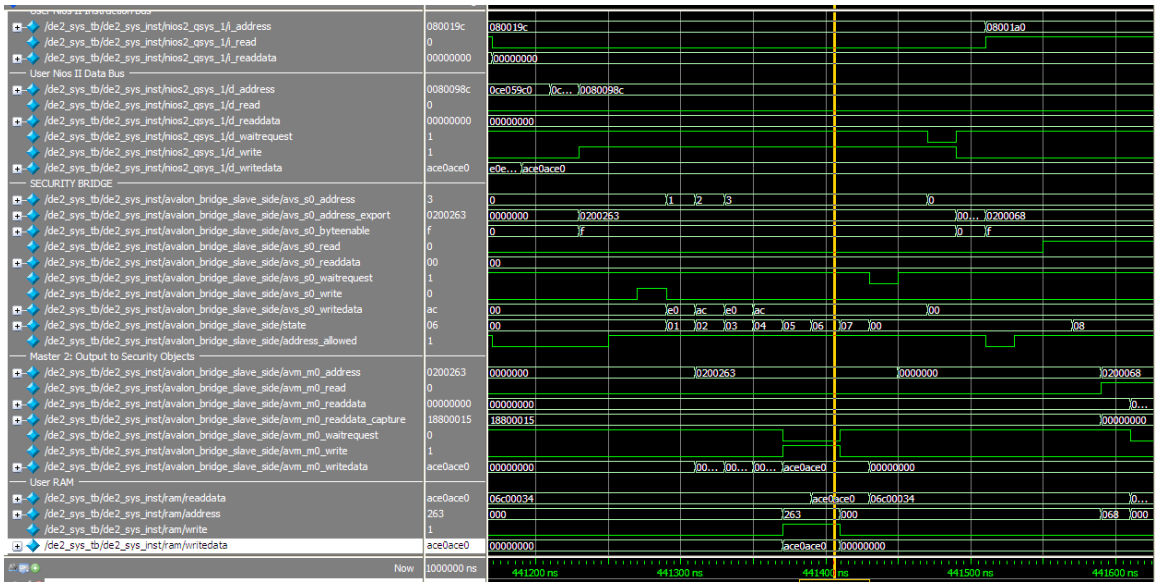

Figure 28: Access Control Test 1 Write

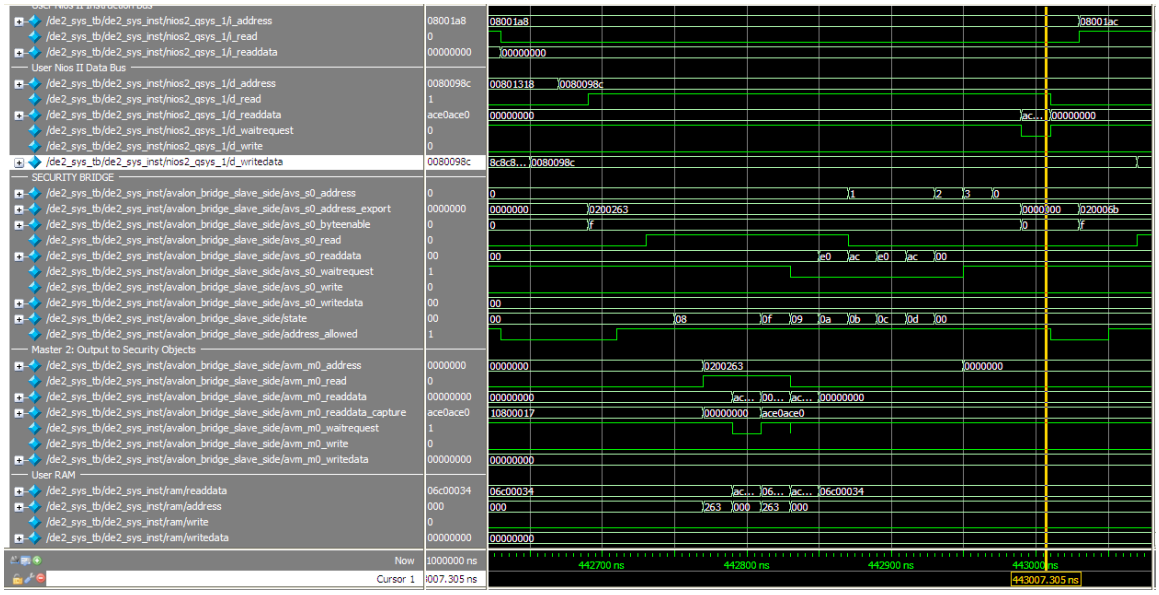

Figure 29: Access Control Test 1 Read

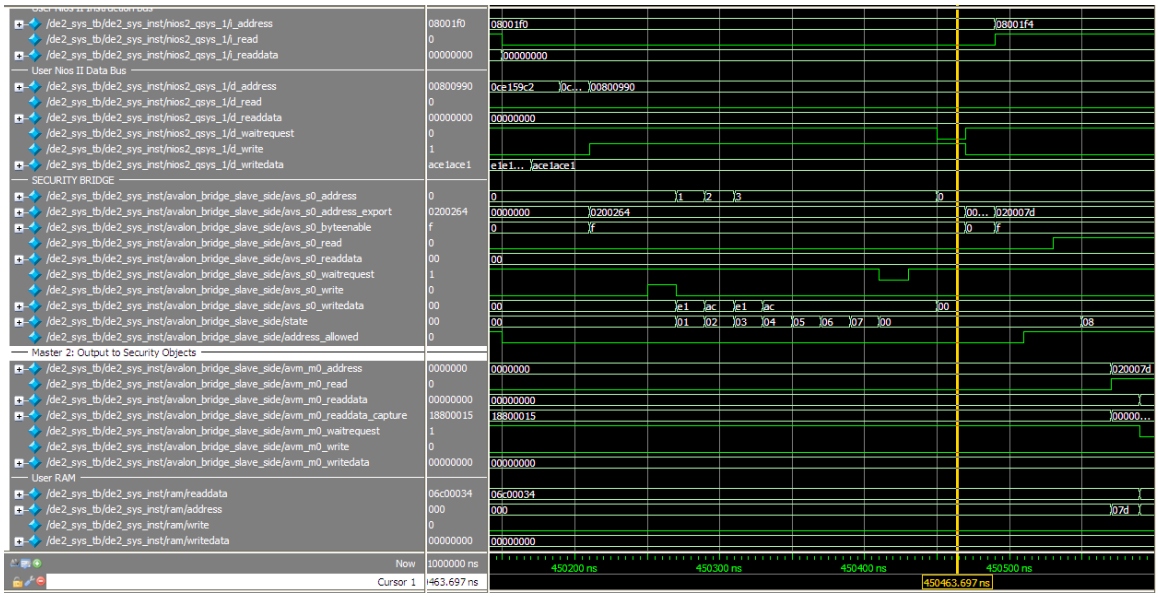

Figure 30: Access Control Test 2 Write

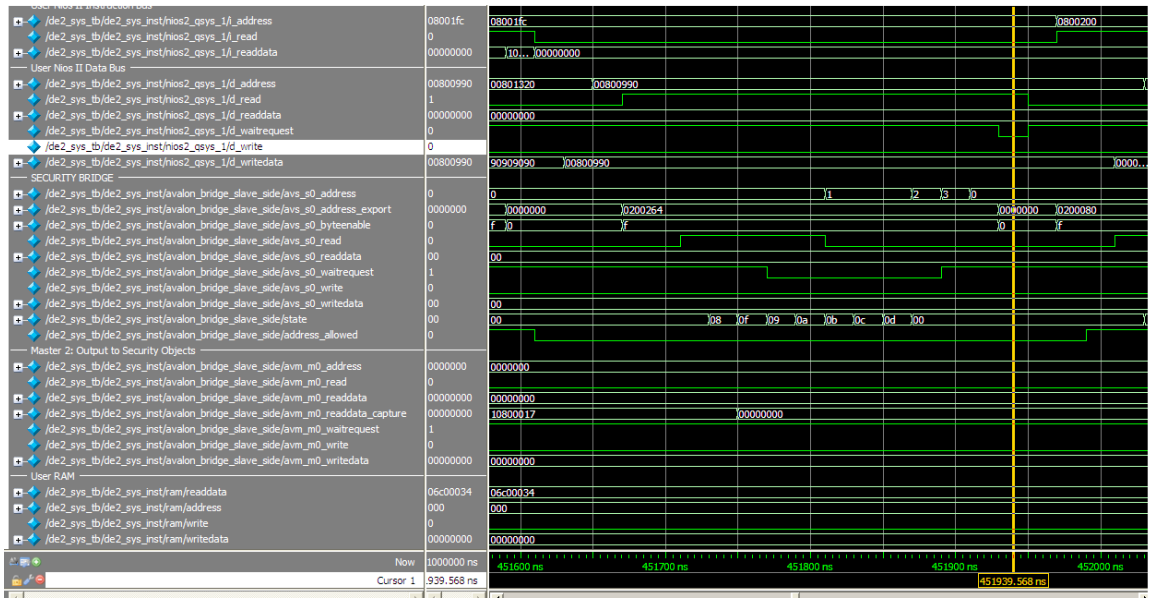

Figure 31: Access Control Test 2 Read

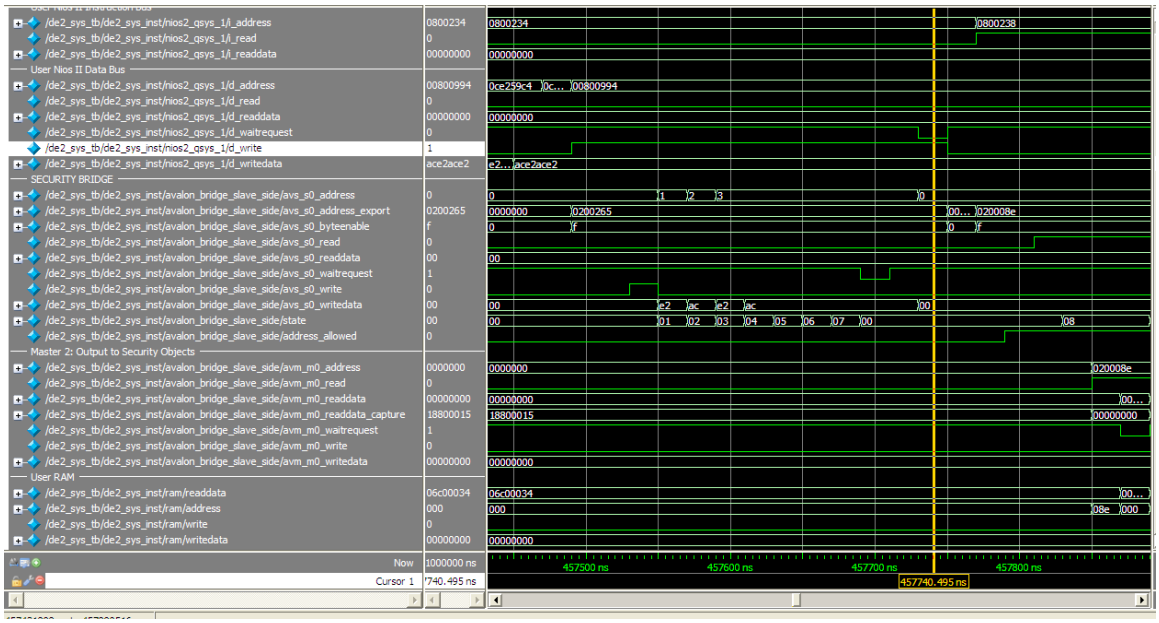

Figure 32. Access Control Test 3 Write

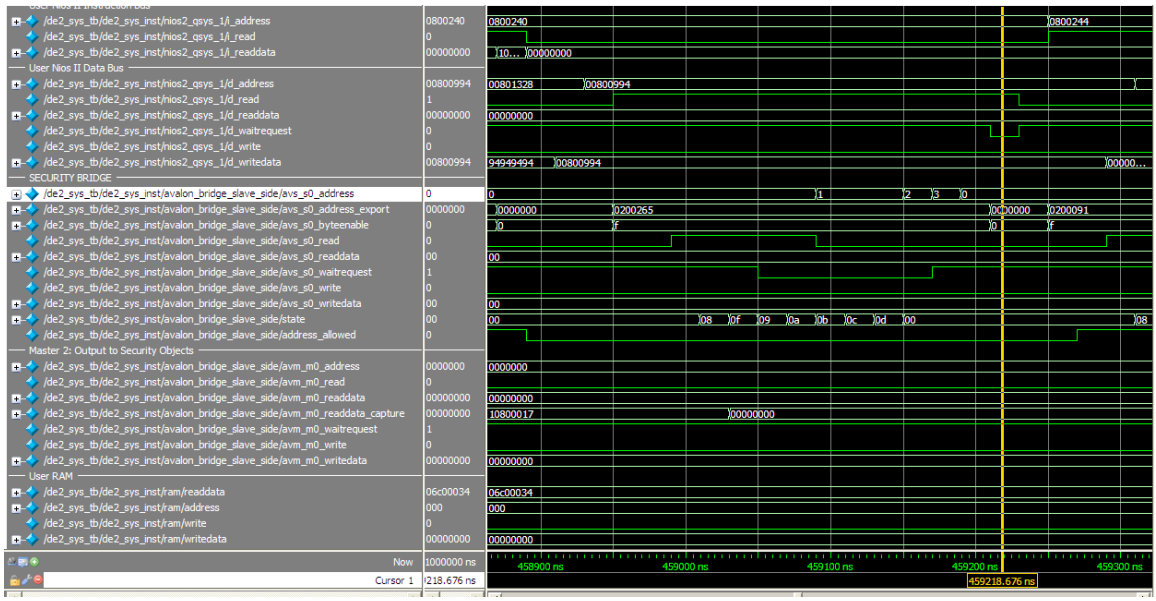

Figure 33. Access Control Test 3 Read

Since this is a demonstration test program, the user Nios II checks that it succeeded with the first operation and that it failed with the next two. The Security Controller ignores invalid access requests and returns zeros for any read data.

In addition, serialization and de-serialization of the data crossing the Security Bridge can be observed in Figures 28-33. Of particular interest is the signal address_allowed. It is the output of a complex logic function in the Security Bridge that evaluates the address requested on each cycle and determines if it is allowed or not. This evaluation is a continuous-class Verilog statement and does not require registering to wait for the next cycle.

Next, operation of the Security Bridge is tested when the security level is increased to level 2. Instructions *80019c, 8001f0,* and *800234* are attempted again. This time, as expected with the access control policy, only the second write/read attempt is allowed. Figures 34-36 show the write operations.
It can be see that test 1 and test 3 have their access attempts rejected.

This process is repeated for level 3. It is verified that only the third test word is allowed to be accessed.

6.2.3 Secure RAM

The Secure RAM is shared between the Security Controller and the User Bridge and allows them to communicate directly. Execution of the custom Verilog to implement this memory is shown in Figure 37. The pass key and the card key are filled in. The user application has requested a checksum of the DE2 Main Board and this is filled in. The user has also requested a reconfiguration operation.

6.3 Secrecy

Figure 38 shows zeroing of the secrets contained on the DE2 Main Board when a tamper event is detected. All keys except the hard-coded board keys are cleared. The security ratchet remains at three as required.

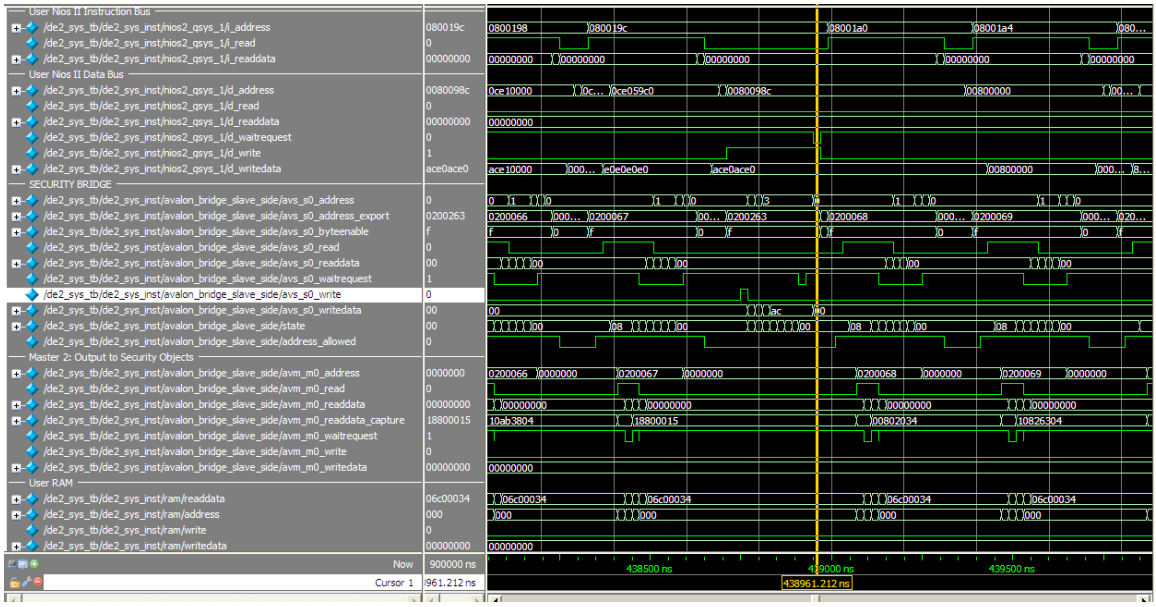

Figure 34. Access Control Test 1, Level 2

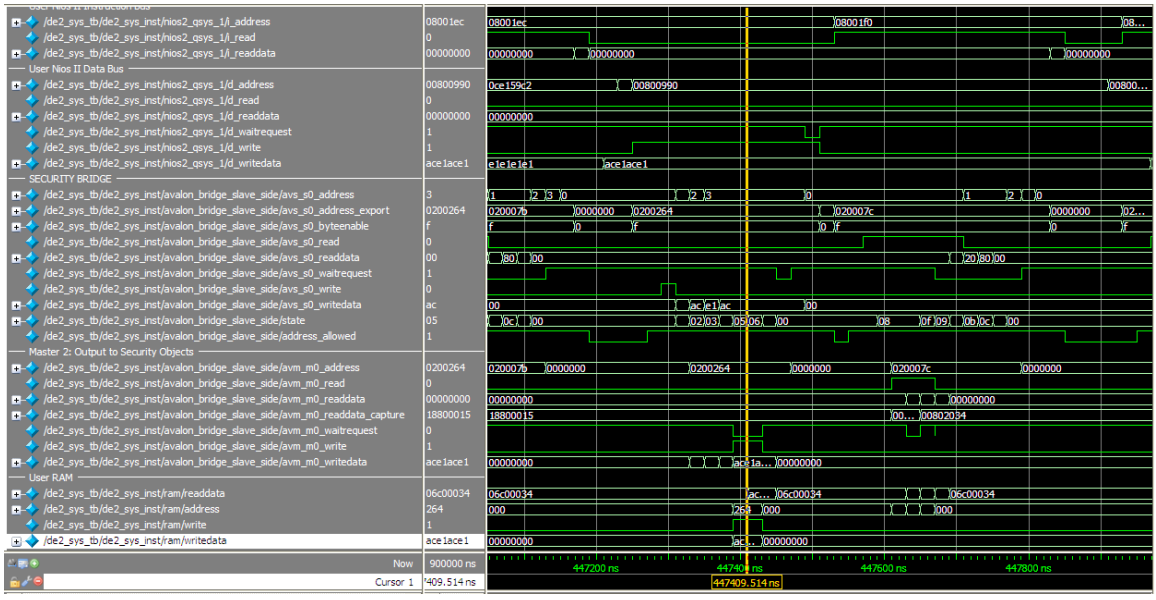

Figure 35. Access Control Test 2, Level 2

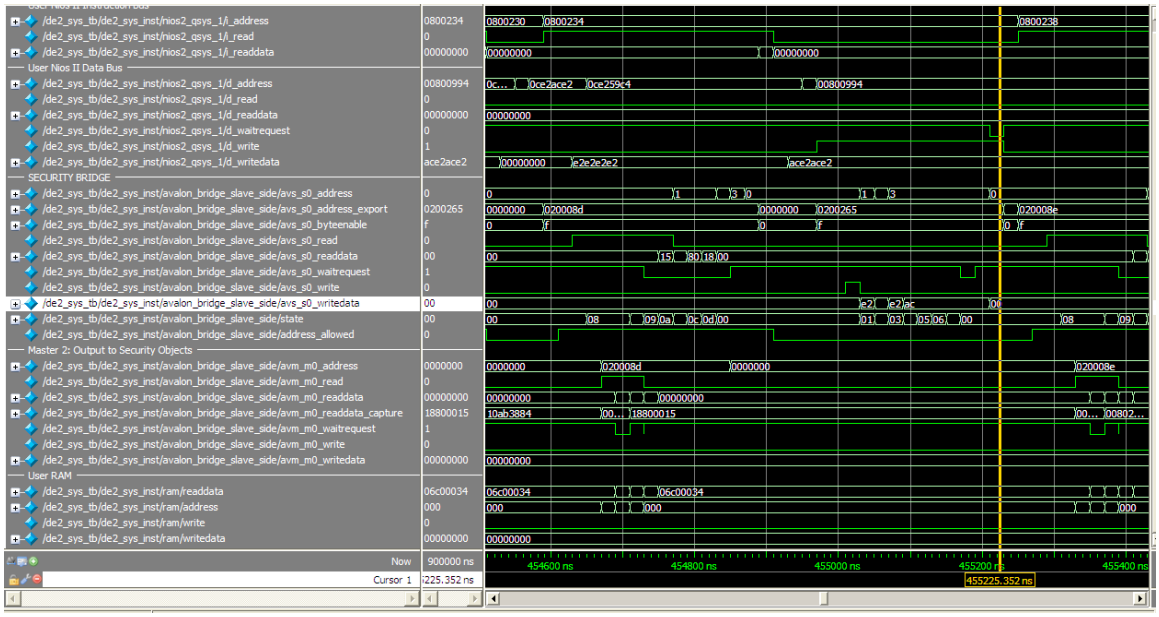

Figure 36. Access Control Test 3, Level 2

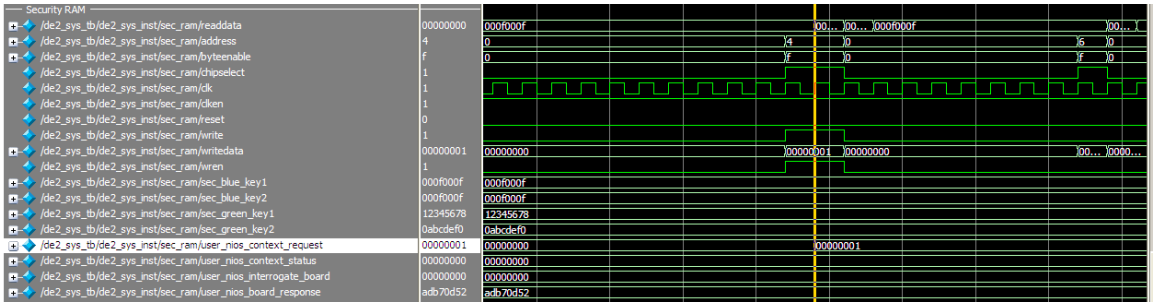

Figure 37. Security RAM Operation

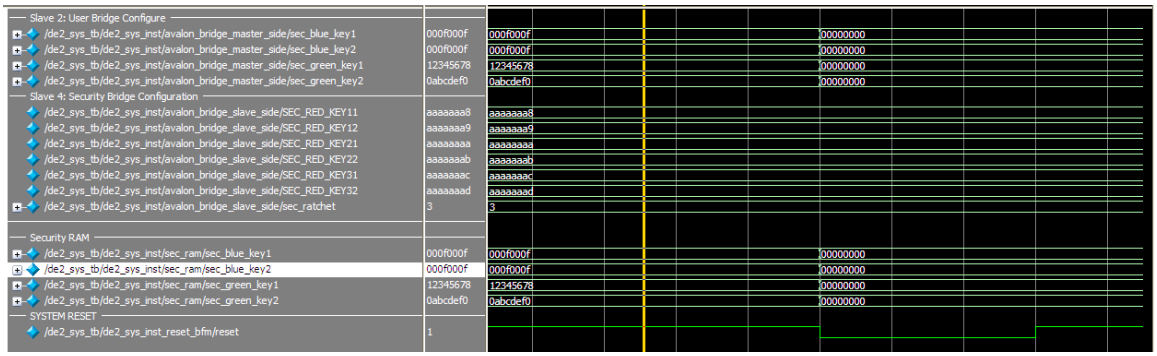

Figure 38. Clearing of Secrets on a Tamper Event

6.4 Authentication

Authentication of the user to the DE2 Main Board is described in section 5.1. The DE2 Main Board accepts the user's pass key from the input terminal and then performs a checksum on it. This checksum is compared to the decrypted checksum from the key card. The system is tested with a bad password and shown to halt:

SC Starting..
..SC HASH 9bc6b732
Testing RAM, Sec Ram, Bridge..
Waiting for SD card..
Checking SD card..
..SD signature 1cf1d9b
..SD calculated signature 1cf1d9b
Enter Password: ght6yj

The second authentication that takes place is the checking that the SD card is signed correctly:

SC Starting..
..SC HASH 9bc6b732
Testing RAM, Sec Ram, Bridge..
Waiting for SD card..
Checking SD card..
..SD signature 1cf1d9b
..SD calculated signature 1cf1daa
..SECURITY FAILURE: halting..

In this case, the security level file is altered:

correct: (startofseclevel)AAA8AAA8AAA8AAA9(endofseclevel)
altered:(startofseclevel)0AA8AAA8AAA8AAA9(endofseclevel)

Note that the signature from the SD card does not match the calculated signature value.

The third authentication is the DE2 Board to the user. This takes place by a checksum of the board that is represented by a checksum of the Security Controller RAM and the state of the DE2 Switches. It is tested by turning on a switch and verifying that that user application halts. This test is completed successfully:

Board Hash 9bc6b733 sent to USER

This value is one more that the correct hash as switch 1 is in the on position. The system halts in an error state according to the user application, which is to indicate $7h$ on the LEDs.

CHAPTER 7
REVIEW AND IMPROVEMENTS

The system described in this thesis meets the design goals in chapter 3. Loss of one part of the cryptosystem, be it the key card, the password, or the DE2 Main Board does not provide an adversary with any information about the key used to encrypt the data in his possession. For the cryptosystem, template functions model ciphers to first order; it is necessary to use cipher-block-chaining. Encrypted code execution is modeled to first order as it is in the electronic code book mode only. Encrypt-then-sign is approximated with the template function. The role based access system is shown clearly through many examples and allows real-time changes to the cycle-level access of the user processor. The secure boot process is demonstrated by the immutable security ratchet counter and attestation of the board state to the user processor. Encrypted data storage and retrieval is shown to be transparent to the user processor once the user processor indicates that the data should be encrypted. The system secrets are shown to clear on a detected tamper event. A first order description of a deployment model is made, in that a system of tools and programs using Scilab are created to act as the toolset that the system owners would use to generate the user files and load them on a key card. Reconfiguration of hardware is demonstrated by two user FPGA configurations, which are initiated as requested by the user application. More specific results in attempting the goals listed in Chapter 2 follow:

- Hardware-enforced RBAC: the user software is shown to have its access to the system limited by the Security Controller based on the predetermined security level. The access control is shown to occur on a cycle-by-cycle basis and can be changed by the Security Controller at any time. This suggests that the system could be extended in an interesting way to work with a compiler system to enforce proper flow control [48]. The user is shown to have to have the key card and the password and this forms a two-part authentication for access. The access policy is shown to be a simple matrix that allows for both overlapping permissions and exclusive permission as it allows entry for every subject versus every object. Software attacks by knowledge insiders would require some type of access violation (this is presupposed by this thesis in that users are granted access to objects or denied access; no checking of the meaning of the communication between the user and the object is checked). The enforcement mechanism is shown to reject the attempted invalid address attempts, so there is no known access control violation.
- Reconfiguration: simple FPGA reconfiguration from a signed, encrypted configuration file residing on the key card is shown. This is interesting in that the key card can be removed before deployment to the field, as long as the user code does not request a context change. This is because the

key card is only used for configuration and can be removed once that is completed. In that mode, a tamper event could clear the entire FPGA and the adversary would have no information about the FPGA. This can be considered in this thesis by assuming a tamper event turns off FPGA power. This concept suggests many useful applications, especially concerning mobile platforms that start in a secure place then leave that place to perform some function.

- Context switching: The system is shown to be able to verify, decrypt, load, and execute two different contexts, with the user code being able to request a context change while running.
- Computation: The system is shown to employ a basic 32-bit microprocessor.
- Secrecy: Consider the threat model in Chapter 3 and Table 3.
    - If the adversary gets the DE2 board while it is running, a tamper event clears the pass key and the card key. All keys could be cleared if the system is implemented with a battery-backed RAM as with the IBM 4758 and IBM 4764. The data on the DE2 is stored (at the user's discretion) with the pass key and the card key. In that manner, it is encrypted as strongly as the cryptosystem allows.
    - If the key card is recovered, the adversary reads it but it is signed and encrypted with the public key for the board, so that data is encrypted as strongly as the cryptosystem allows. Changing the card requires forging the CA signature, which is designed by the cryptosystem to be hard.
    - Recovering the key card and the DE2 board gives the adversary all of the information on the key card as the board secret key is compromised. This is the FPGA files, the application files, the security level file, the card key file, and the password hash file. However, data that the user wishes to have secondary encryption applied to is encrypted still with the pass key half. It is reviewed that changing the system from concatenating the card key with the pass key to performing the exclusive-or mixing of them would leave the data fully encrypted, but with the concatenation scheme, the number of bits is halved.
    - Recovering the password and the DE2 board gives the adversary the board secret key. However, the user data is encrypted with the key card key and the pass key. Again, possessing the pass key reduces the security parameter bits by half for the concatenated key system used in this thesis, but an exclusive-or implementation would leave the data fully encrypted.
- Trust: The system models a secure boot in that it performs system tests at start up and halts on a problem and it performs a measurement of the system and can provide that to the user application on demand. It employs a monotonic counter than can only be reset by a full system power-off and reprogramming.

- SoPC: The system is designed on a highly configurable system-on-programmable-chip fabric.
- Tamper reaction: Tamper reaction is shown to clear the secrets on the DE2 board. The SD card does not employ tamper reaction, but certainly could in an active token design. In addition, by inspection it can be said that the FPGA power switch is the tamper switch, in which case all information about the FPGA and user applications is cleared. The only remaining data would be data stored by the user in a flash memory (not implemented, but modeled). This data would be fully encrypted with the card key and the pass key.
- Authentication: To use the system, the user is demonstrated to have to use the key card and the password. The Security Controller is shown to attest its configuration to the user software, which can decide if it is acceptable or not. The system is shown to be assured of the identity of the key card in as much as the signature cryptosystem is secure.
- User-initiated data secrecy: The system is shown to allow the user to apply secondary encryption to instruction and data using the pass key and the card key. In this manner, the encryption of the user data is separate from the board key, whose purpose is to receive configuration information from the key card. The system is shown to allow the user software to enable or disable this encryption function at any time.
- Deployment: A simple model for system deployment is provided including how encrypted data gets to the system and its source. In addition, the expectation of the system is noted in the threat model presented.

To best use the concepts presented in this thesis, the following recommendations are made:

1. Change the system to a single-chip solution to increase speed.
2. In the single-chip solution, use wide serial buses to connect the User Bridge to the Security Bridge and remove the SERDES.
3. Redesign the User Bridge so that the toolset can understand that it is a bridge to secondary peripherals, or use an SoPC system that recognizes this.
4. Create custom applications to automatically generate and encrypt the files and to load them to the key card.
5. Make the key card active and have its secrets zero on tamper. Also, make it respond to command-response tests for identification.
6. Possibly use biometrics for identification.
7. Enable the use of interrupts and other full-processor features.
8. Use a development platform that has up-to-date IP already made for the peripherals instead of one that does not.
9. Expand the number and types of peripherals available to the user.
10. Use a stream cipher for instruction encryption.
11. Provide a tamper-detecting enclosure.

12. Use as many bits as possible for the encryption system.
13. Allow for system updates.
14. Allow for a full operating system.
15. Tag information with level numbers and control the information flow.
16. Create a compiler system that can take advantage of hardware that allows dynamic access control to increase security.
17. Allow programs to be stored in flash.
18. Increase the number of levels to four: unclassified, classified, secret, top secret.
19. Use a more powerful processor.
20. Increase the signature length.
21. Clear the FPGA configuration on a tamper event.
22. Change the checksum to a cryptographic hash.

LIST OF REFERENCES

[1] J. K. Katz and Y. Lindell, Introduction to Modern Cryptography: Principles and Protocols, Chapman and Hall/CRC, 2007.

[2] F. Bauer, "Cryptology," *Encyclopedia of Cryptology and Security,* p. 118, 2005.

[3] F. L. Bauer, "Cryptanalysis," *Encyclopedia of Cryptology and Security,* pp. 113-114, 2005.

[4] C. E. Shannon, "Communication theory and secrecy systems," *Bell Systems Technical Journal,* vol. 28, pp. 656-715, 1949.

[5] F. L. Bauer, "Encryption," *Encyclopedia of Cryptology and Security,* p. 202, 2005.

[6] H. van Tilborg, "Shannon's Model," *Encyclopedia of Cryptography and Security,* p. 568, 2005.

[7] F. Bauer, "Cryptosystem," *Encyclopedia of Cryptology and Security,* p. 119, 2005.

[8] L. R. Knudsen, "Block Ciphers," *Encyclopedia of Cryptology and Security,* pp. 41-46, 2005.

[9] K. Sako, "Public Key Cryptography," *Encyclopedia of Cryptology and Security,* pp. 487-488, 2005.

[10] B. Preneel, "Modes of Operation of a Block Cipher," *Encyclopedia of Cryptology and Security,* p. 386, 2005.

[11] J. Daemen and V. Rijmen, "Rijndael / AES," *Encyclopedia of Cryptology and Security,* pp. 520-524, 2005.

[12] A. Biryukov and C. De Canniere, "Data Encryption Standard," *Encyclopedia of Cryptology and Security,* pp. 129-134, 2005.

[13] A. Canteaut, "Stream Cipher," *Encyclopedia of Cryptology and Security,* pp. 596-597, 2005.

[14] A. Canteaut, "A5/1," *Encyclopedia of Cryptology and Security,* p. 1, 2005.

[15] C. Adams, "Replay Attack," *Encyclopedia of Cryptology and Security,* p. 519, 2005.

[16] B. Preneel, "Hash Functions," *Encyclopedia of Cryptology and Security,* pp. 257-264, 2005.

[17] B. Preneel, "MAC Algorithms," *Encyclopedia of Cryptology and Security,* pp. 361-368, 2005.

[18] J. Black, "Authenticated Encryption," *Encyclopedia of Cryptology and Security,* pp. 11-20, 2005.

[19] J. An, Y. Dodis and T. Rabin, "On the Security of Joint Signature and Encryption," *2002.*

[20] M. Just, "Challenge-Response Identification," *Encyclopedia of Cryptology and Security,* pp. 73-74, 2005.

[21] R. Zuccherato, "Identity Verification Protocol," *Encyclopedia of Cryptology and Security,* pp. 285-286, 2005.

[22] R. Zuccherato, "Entity Authentication," *Encyclopedia of Cryptology and Security,* p. 203, 2005.

[23] A. Jain and A. Ross, "Biometrics," *Encyclopedia of Cryptology and Security,* pp. 34-36, 2005.

[24] G. Brose, "Access Control," *Encyclopedia of Cryptology and Security,* pp. 2-6, 2005.

[25] C. Adams, "Authorization Architecture," *Encyclopedia of Cryptology and Security,* pp. 23-27, 2005.

[26] S. W. Smith, Trusted Computing Platforms: Design and Applications, Boston: Springer Science + Business Media, Inc., 2005.

[27] M. Wiener, "Exhaustive Key Search," *Encyclopedia of Cryptology and Security,* pp. 206-209, 2005.

[28] C. Adams, "Dictionary Attack I," *Encyclopedia of Cryptology and Security,* p. 147, 2005.

[29] E. Cronin, "Denial of Service," *Encyclopedia of Cryptology and Security,* pp. 143-144, 2005.

[30] A. Biryukov, "Code Book Attack," *Encyclopedia of Cryptology and Security,* p. 80, 2005.

[31] Y. Desmedt, "Man-in-the-Middle Attack," *Encyclopedia of Cryptology and Security,* p. 368, 2005.

[32] C. Adams, "Impersonation Attack," *Encyclopedia of Cryptography and Security,* p. 286, 2005.

[33] A. Biryukov, "Adaptively Chosen Ciphertext Attack," *Encyclopedia of Cryptology and Security,* p. 7, 2005.

[34] A. Biryukov, "Adaptive Chosen Plaintext Attack," *Encyclopedia of Cryptography and Security,* p. 8, 2005.

[35] A. Biryukov, "Adaptive Chosen Plaintext and Chosen Ciphertext," *Encyclopedia of Cryptography and Security,* p. 7, 2005.

[36] A. Biryukov and C. De Canniere, "Linear Cryptanalysis for Block Ciphers," *Encyclopedia of Cryptology and Security,* pp. 351-353, 2005.

[37] E. Biham, "Differential Cryptanalysis," *Encyclopedia of Cryptology and Security,* pp. 147-148, 2005.

[38] J.-J. Quisquater and S. David, "Electromagnetic Attack," *Encyclopedia of Cryptology and Security,* pp. 170-172, 2005.

[39] O. Benoit, "Fault Attack," *Encyclopedia of Cryptology and Security,* pp. 218-219, 2005.

[40] M. Kuhn, "Data Remanence," *Encyclopedia of Cryptology and Security,* p. 135, 2005.

[41] Altera, *Cyclone II Device Handbook, Volume I,* San Jose, CA, 2008.

[42] Altera, Quartus II Handbook Version 13, San Jose: 2013.

[43] Altera, *Cyclone III Device Handbook,* San Jose, 2012.

[44] R. Sandhu, E. Coyne, H. Feinstein and C. Youman, "Role-Based Access Control Modes," *IEEE Computer,* vol. 29, no. 2, pp. 38-47, 1996.

[45] D. Ferraiolo and D. Kuhn, "Role Based Access Control," in *15th National Computer Security Conference*, Baltimore, 1992.

[46] NIST, "Role Based Access Control - Frequently Asked Questions," [Online]. Available: http://csrc.nist.gov/groups/SNS/rbac/faq.html. [Accessed 11 July 2013].

[47] R. Sandhu, E. Coyne, H. Feinstein and C. Youman, "Role-Based Access Control: A Multi-Dimensional View*".

[48] J. Franklin, S. Chaki, A. Datta, M. J. McCune and A. Vasudevan, "Parametric Verification of Address Space Separation".

[49] K. Zhang, T. Zhang and S. Pande, "Memory Protection through Dynamic Access Control".

[50] Z. Zhou, V. Gilgor, J. Newsome and J. McCune, "Building Verifiable Trusted Path on Commodity x86 Computers".

[51] A. Triulzi, "The Jedi Packet takes over the Deathstar, taking NIC backdoor to the next level.," in *The 12th Annual CanSec West Conference*, 2010.

[52] ARM, *ARM Security Technology: Building a Secure System using TrustZone Technology,* 2009.

[53] M. Gasser, A. Goldstein, C. Kaufman and B. Lampson, "The Digital Distributed System Security Architecture," in *Proceedings of the National Computer Security Conference*, 1989.

[54] A. Martin, "The ten-page introduction to Trusted Computing," 2008.

[55] B. Glas, A. Klimm, K. Muller-Glaser and J. Becker, "Configuration Measurement for FPGA-based Trusted Platforms," 2009.

[56] B. Parno, J. McCune and A. Perrig, "Bootstraping Trust in Commodity Computers".

[57] B. Glas, A. Klimm, D. Schwab, K. Muller-Glaser and J. Becker, "A prototype of trusted platform functionality on reconfigurable hardware for bitstream updates," 2008.

[58] T. Caddy, "FIPS 140-2," *Encyclopedia of Cryptology and Security,* pp. 227-230, 2005.

[59] IBM, "IBM 4764 PCI-X Cryptographic Coprocessor," [Online]. Available: http://www-03.ibm.com/security/cryptocards/pcixcc/overhardware.shtml. [Accessed 15 July 2013].

[60] P. Chu, Embedded SoPC Design with Nios II Processor and Verilog Examples, Wiley, 2012.

[61] P. Chu, FPGA Prototyping by Verilog Examples, Wiley, 2008.

[62] Altera, *Avalon Interface Specifications,* San Jose, CA, 2013.

[63] Mentor Graphics, "ModelSim PE," [Online]. Available: http://modelsim.s3.amazonaws.com/modelsim-pe-datasheet.pdf. [Accessed 25 July 2013].

[64] Altera, "Nios II Processor: The World's Most Versatile Embedded Processor," [Online]. Available: http://www.altera.com/devices/processor/nios2/ni2-index.html. [Accessed 25 July 2013].

[65] IEEE, "1149.7-2009 - IEEE Standard for Reduced-Pin and Enhanced-Functionality Test Access Port and Boundary-Scan Architecture," 2010.

[66] SanDisk, "SanDisk SD and SDHC Cards," [Online]. Available: http://www.sandisk.com/products/memory-cards/sd/standard-class4/?capacity=2GB. [Accessed 25 July 2013].

[67] MH- Nexus, "HxD - Freeware Hex Editor and Disk Editor," [Online]. Available: http://mh-nexus.de/en/hxd/. [Accessed 25 July 2013].

[68] L. Jasio, Programming 32-bit Microcontrollers in C: Exploring the PIC32, Newnes, 2008.

[69] Eclipse, "Eclipse," [Online]. Available: http://www.eclipse.org/. [Accessed 25 July 2013].

[71] G. Kabatiansky and B. Smeets, "Authentication," *Encyclopedia of Cryptology and Security,* pp. 21-22, 2005.

[72] F. Bauer, Encyclopedia of Cryptography and Security, 2005.

VITA

William Collins graduated from the University of Tennessee, Knoxville with a BSEE in 1999 and has developed applications across military, industrial, and research domains employing digital, analog, and power systems design.