Securing Software Intellectual Property on Commodity and Legacy Embedded Systems

Michael A. Gora

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

Master of Science
in
Computer Engineering

Patrick R. Schaumont, Chair
Joseph G. Tront
Randolph C. Marchany

June 1st, 2010
Blacksburg, Virginia

Keywords: Secure Embedded Systems, Software, Intellectual Property, Firmware, Security, FPGA, Design Flow, Physical Unclonable Function

Copyright 2010, Michael A. Gora
Securing Software Intellectual Property on Commodity and Legacy Embedded Systems

by

Michael A. Gora

Abstract

The proliferation of embedded systems into nearly every aspect of modern infrastructure and society has seen their deployment in such diverse roles as monitoring the power grid and processing commercial payments. Software intellectual property (SWIP) is a critical component of these increasingly complex systems and represents a significant investment to its developers. However, deeply immersed in their environment, embedded systems are difficult to secure. As a result, developers want to ensure that their SWIP is protected from being reverse engineered or stolen by unauthorized parties. Many techniques have been proposed to address the issue of SWIP protection for embedded systems. These range from secure memory components to complete shifts in processor architectures. While powerful, these approaches often require the development of systems from the ground up or the application of specialized and often expensive hardware components. As a result they are poorly suited to address the security concerns of legacy embedded systems or systems based on commodity components. This work explores the protection of SWIP on heavily constrained, legacy and commodity embedded systems. We accomplish this by evaluating a generic embedded system to identify the security concerns in the context of SWIP protection. The evaluation is applied to determine the limitations of a software only approach on a real world legacy embedded system that lacks any specialized security hardware features. We improve upon this system by developing a prototype system using only commodity components. Finally we propose a Portable Embedded Software Intellectual Property Security (PESIPS) system that can easily be deployed as a framework on both legacy and commodity systems.
Acknowledgements

Through the course of this work and my college career I have received a great deal of support from a great deal of people. I would like to directly thank the following people:

- My parents, Marian and Margaret, for supporting me throughout my life and helping me make decisions that have benefited me deeply. I would not be a Computer Engineer or here today if it wasn’t for their hard work and sacrifice and I hope that I can continue to make them as proud of me as I am of them.

- My advisor, Dr. Patrick Schaumont, for being a source of guidance in my college career and providing me with so many opportunities to expand myself not only academically but personally as well. He is truly one of the hardest working and dedicated individuals I have met in my life and I think him deeply for giving me input and a push in the right direction when I needed it.

- My committee member, Dr. Joseph Tront, for not only providing support and flexibility in graduate career but providing opportunities that have helped shape my academic and future career. A more insightful and personable instructor you could not ask for and he has made my experience at Virginia Tech all the more memorable.

- My committee member, Mr. Randolph Marchany, a renowned professional in the security world, talented musician, and in no small part responsible for where I am today, I am grateful to have met and worked with him.

- My friend, Colonel Dr. Timothy Buennemeyer, for giving me my initial break into research and the world of security but for providing me with all the opportunities that our collaboration yielded. It was his support that encouraged me to continue on to obtain my Masters Degree and I cannot thank him enough for it.

- My friend, Lee M. Clagett II, for being a good friend for over seven years now, having introduced me to Colonel Dr. Buennemeyer and Mr. Marchany, and getting me started in research. Life is always more interesting when you surround yourself with interesting and intelligent people, Lee has always been proof of this.
• My loved ones and friends, Lauren Hefty, Tannus Frangieh, Sarah Mac, and so many more, for being there to support me even when times became difficult and I was no longer able to be there as much as I like. Thank you so much, I will always cherish our relationships and look forward to continuing them where ever life may lead.

• My coworkers over the years, Abhranil Maiti, Xu Guo, Zhimin Chen, Sergey Morozov, John Paul Dunning, Michael Henry, Theresa Nelson Robinson, and many more, for being sources of knowledge and expertise that have helped me grow as a student and professional.

• The faculty and staff of Virginia Tech, Instruct Jason Thweatt, Dr. Tom Martin, Dr. Eunice Santos, Dr. Ezra Brown, Dr. Matthew R. Goodrum, Miss. Mary Taylor, and Miss. Becky Semones, who have influenced and provided aid for my academic and professional careers. Thank you for making my time at Virginia Tech so memorable.

• Finally, to everyone in my life who I may have not mentioned, thank you for your love, support, and help.
6 Portable Embedded Software Intellectual Property Security
   6.1 Overlap in Securing SWIP on Embedded Systems .............................. 68
   6.2 PESIPS System Overview .......................................................... 68
   6.3 Intellectual Property Encryptor .................................................... 70
      6.3.1 Generating Configuration Files ............................................. 72
      6.3.2 Encryption and Configuration ................................................ 74
   6.4 PESIPS Module ............................................................................. 76
      6.4.1 Integrity Kernel ...................................................................... 77
      6.4.2 Security Kernel ...................................................................... 77
      6.4.3 Component Management .......................................................... 79
   6.5 PESIPS Requirements ................................................................. 79
      6.5.1 Shared Cryptographic Library .................................................. 79
      6.5.2 Application of Naming Conventions ........................................... 83
      6.5.3 Trusted Boot ........................................................................... 83
      6.5.4 System Storage Integration ...................................................... 83
      6.5.5 PESIPS Module Configuration .................................................. 84
   6.6 PESIPS and the Trusted Platform Module ......................................... 85
   6.7 Summary ....................................................................................... 86

7 Results ............................................................................................... 87
   7.1 Overview ....................................................................................... 87
   7.2 Legacy Embedded System Implementation ...................................... 87
      7.2.1 Demonstration System .............................................................. 87
      7.2.2 Software Utilization ................................................................. 88
      7.2.3 System Performance ................................................................. 88
   7.3 Commodity Embedded System Implementation .............................. 89
      7.3.1 Demonstration System Hardware Architecture .......................... 89
      7.3.2 Physical Unclonable Function (PUF) .......................................... 90
7.3.3 Functionality Testing ........................................ 90
7.3.4 PUF Characterization ........................................ 90
7.3.5 Hardware Utilization ......................................... 91
7.3.6 Software Utilization .......................................... 91
7.3.7 System Performance .......................................... 93

7.4 PESIPS Implementation ......................................... 93
7.4.1 Software Utilization .......................................... 93

7.5 Summary ......................................................... 93

8 Conclusion ......................................................... 95
8.1 Contributions ................................................... 95
8.2 Future Work ..................................................... 96

Bibliography ......................................................... 97
Tables

Table

3.1 Messages transmitted during ECDH Key Exchange and distribution of SWIP key. ......................................................... 29
4.1 Messages exchanged for DS2432 based authentication. ......................... 43
4.2 Authenticated ECDH protocol exchange details. ................................. 45
7.1 Legacy System Solution Memory Utilization .................................. 88
7.2 PUF Metrics ........................................................................... 91
7.3 Commodity Solution Resource Utilization Xilinx Spartan XC3500E .... 91
7.4 Commodity System Solution Memory Utilization ................................. 92
7.5 PESIPS System Solution Memory Utilization ................................... 94
Figures

Figure

2.1 The typical hardware configuration for an embedded sensor node. Dashed regions represent functional units of the system and shaded components are those which SWIP is present on. ................................................................. 5
2.2 The architecture associated with the core components of a legacy embedded system that are exposed to SWIP. ................................................................. 6
2.3 Trust boundary and relationships between system components and development parties. Dashed regions which overlap signify trusted relationship between parties. ................................................................. 10
2.4 System design flow from hardware development through end use, shaded regions highlight primary focus of work. ................................................................. 10
2.5 System design flow from end of system development through the creation of the final embedded system product. ................................................................. 12
2.6 System design flow from end of system development through the creation of the final embedded system product. ................................................................. 13
3.1 Modified software development design flow to support SWIP protection. Shaded regions denote additions to the design flow. ................................................................. 21
3.2 Encryption of an Elf binary by the IPE. ................................................................. 24
3.3 Decryption of software sections by the SK at boot. ................................................................. 24
3.4 Block cipher, such as AES, basic mode of operation known as ECB. ................................................................. 25
3.5 Counter mode of operation for a block cipher. ................................................................. 26
3.6 Establishing a confidential channel utilizing ECDH key exchange and AES. ................................................................. 28
3.7 Elliptic Curve Diffie-Hellman public key exchange. ................................................................. 30
3.8 Davies Mayer hash construction using a block cipher. ................................................................. 32
3.9 Design flow overview, shaded blocks are additions or modifications. ................................................................. 33
4.1 Example of Man in the Middle Attack on a simplified version of our SWIP key exchange protocol.

4.2 DS2432 overview, write MAC is verified by SHA-1 engine and read MAC is generated by the SHA-1 engine.

4.3 Data utilization to generate DS2432 write MAC.

4.4 Data utilization to generate a DS2432 read MAC.

4.5 Authentication overview.

4.6 DS2432 Authenticated ECDH unified authentication and key exchange protocol.

5.1 SWIP binding to a platform comprised of HWIP and an FPGA.

5.2 The architecture associated with a FPGA based commodity embedded system.

5.3 Trust boundary and relationships between system components and development parties. Dashed regions which overlap signify trusted relationship between parties.

5.4 System design flow from hardware development through end use, shaded regions highlight primary focus of work.

5.5 Modified software development design flow to support SWIP protection. Shaded regions denote additions to the design flow.

5.6 Encryption of an Elf binary by the IPE.

5.7 Decryption of software sections by the SK at boot.

5.8 Trusted boot procedure through IK verification of the SK.

5.9 Design flow overview. Shaded blocks represent additions or modifications.

6.1 SWIP security design flow overview for legacy and commodity systems.

6.2 Generic design flow for deployment of the PESIPS system. Shaded figures denote PESIPS system components.

6.3 Iterative approach to the encryption and configuration of ELF binary by PESIPS system IPE.

6.4 PESIPS based encryption of a binary file.

6.5 PESIPS module deployed onto embedded system to authenticate and decrypt SWIP at boot time.

6.6 Trusted boot procedure through IK verification of the SK.

6.7 PESIPS decryption of an encrypted system through the security kernel, note the developer provided system storage.
7.1 SWIP binding prototype system architecture on a Xilinx Spartan XC3500E FPGA
Chapter 1

Introduction

The proliferation of embedded systems into nearly every aspect of modern infrastructure and society has seen their deployment in increasingly demanding applications. Such roles range from the trivial: a thermostat monitoring the temperature of a home, to the safety critical: the avionics controlling an airplane. In parallel with these ever more demanding applications and their widespread adaptation the complexity of embedded systems has grown as well. As a result developers are increasingly spending more time, technical ability, and capital on their development. Products containing innovative proprietary technology, referred to as intellectual property (IP), represent a substantial financial benefit to developers who can bring them to an unsaturated market. Many developers rely primarily on these initial sales of their product to recoup their investments [45]. However, success often brings theft which can erode returns, significantly impacting profitability. This effect is even more prominent when an unscrupulous competitor pirates or reverse engineers a product, allowing them to sustain significantly lower development costs. Not only do such imitations undermine the market position of a developer but they open up possibly application sensitive systems to attack. As a result, developers need to protect their IP from reverse engineering, counterfeiting, and tampering. What is needed is a solution that can:

- Guarantee the confidentiality of SWIP to prevent its theft.
- Insure the integrity of system software to prevent malicious modification.
- Limit the use of SWIP only to authentic customers and system.
- Ensure that authenticity of SWIP running on a system.

This is highlighted by the estimated 5%-10% of high technology products on the market are counterfeit [28, 63, 36], a trend which continues to increase [35, 50]. Software intellectual property (SWIP) is a critical and complex component of embedded systems. To maximize the return on its development, SWIP is often designed to be portable and easily upgraded, allowing rapid reuse and expansion across a variety of platforms. This naturally makes SWIP a prime target for theft as it is easily reverse engineered and redeployed. Even worse in the case of critically important networks, reverse engineering could aid in the disruption of service or theft of sensitive data. A common fear is that an unsecured product, discarded
or stolen, appears on the black market where it can be obtained by a competitor. In order to protect their investment, it is the goal of the developer to make the process of reverse engineering SWIP costly enough to discourage any such attempts. Protecting SWIP from the threat of piracy is a long standing concern for developers of traditional PC systems. Various techniques have been proposed in this field such as obfuscation [25], software encryption, and specialized cryptographic hardware [27]. However these approaches are not always suited to resource constrained embedded systems which offer an order of magnitude less computational power, storage, and energy. More importantly, embedded systems are deeply immersed in their environment, exposing them to direct physical attack. Traditional PC based security models generally do not account for this type of adversarial challenge. Protecting SWIP on embedded systems is a relatively new trend that stems from original efforts to improve tamper resistance. These include physical security such as potting or secure enclosures [21], secure cryptographic components [7], and custom cryptographic architectures [42]. However these solutions lend themselves to hardware based approaches which leave a great deal of legacy systems vulnerable as hardware retrofits are too costly of infeasible. Even on newly developed systems, such components may be too expensive or inflexible to suit a particular application. This work seeks to addresses the issue of providing security for SWIP on legacy and commodity systems with a minimum of modification or specialized components. Our approach is organized as follows:

- In Chapter 2 we begin by presenting a typical legacy embedded system, such as a sensor node. An evaluation is performed on the example system, creating a security model that identifies key vulnerabilities. This is followed by a brief literature survey to analyze any related work or existing solutions in the context of SWIP protection for legacy of commodity systems. Through the security model and literature survey we are able to identify "Software Encryption" as one of the more flexible and practical candidates for deterring theft or reverse engineering of SWIP.

- In Chapter 3, based on our evaluations we propose and implement a prototype system for encrypting SWIP on the example legacy embedded system. An evaluation is then performed on the implementation to identify any limitations of our prototype resulting from the nature of its target system.

- In Chapter 4 we explore two such critical flaws that were identified; no secure storage for encryption keys and the lack of an authentication mechanism for the device. We iteratively address these issues, performing security evaluations and determining possible solutions until we are able to afford a degree of security with only minimal hardware modification.

- In Chapter 5, the prototype SWIP protection system developed for the legacy system serves as an important learning tool. Using it as a model we identify current commodity components that can be used to address many of the limitations of its legacy hardware without relying on specialized components. Specifically we identify systems based on commodity Field Programmable Gate Arrays (FPGA) as excellent candidates due to their ability to provide a self contained System on Chip
(SOC) environment. Additionally, they provide several features with possible security implications, such as chip unique physical identification through the use of Physical Unclonable Functions (PUFs), without the need for specialized hardware. We exploit these features to simplify and increase the security of our original SWIP protection prototype system.

- In Chapter 6, we evaluate the many similar components that our SWIP protection systems possess despite being based on different hardware and usage scenarios. Only slight implementation details vary such as the mechanism used to store the SWIP encryption keys. Based on this observation we propose a Portable Embedded Software Intellectual Property Security (PESIPS) system to facilitate in the rapid deployment of SWIP protection on a broad array of embedded systems.

- In Chapter 7, we present metrics from the implementation of our legacy, commodity, and generic PESIPS implementations.

- In Chapter 8, we conclude with a brief evaluation of the impact of each of our systems and the key lessons we derived by carrying them through design into implementation.
A critical first step in developing a protection mechanism for embedded SWIP is to understand the platform on which it is deployed. In this section we present a typical legacy embedded system which we use as a basis for developing our target system model. We expand on this through a detailed security evaluation that identifies possible system weaknesses. The section culminates with a brief literature survey to identify any existing techniques or readily available solutions to address any identified critical security issues.

2.1 Legacy System

The term legacy embedded system doesn’t imply a system with a particular purpose or hardware arrangement. Rather, in the context of this work, we refer to a legacy embedded system as one which possesses the following properties:

- The system’s hardware design is fixed and not subject to major revision.
- The system is currently in production and/or deployed.
- The system does not possess any specialized secure or cryptographic hardware components.
- The system may have basic tamper detection mechanisms.
- The system may have basic physical protection mechanisms.

Under such broad classification, the majority of existing system designs could be considered legacy embedded systems. However, the time associated with addressing the near endless permutations of system components, involved parties, and trust models covered under this label would be prohibitive. Instead we choose to focus on the core components of a typical legacy embedded system, such as an embedded sensor node.
2.2 Legacy Embedded System Architecture

The nodes found in an embedded sensor network offer the ideal target for our evaluation. Not only do they possess the core components of typical legacy embedded systems, but they benefit greatly from SWIP protection. This is due to the often complex and highly optimized SWIP deployed on the devices combined with their use in unsecured environments. Let us take into consideration the architecture of a generic sensor node, as depicted in Figure 2.1, and divide its hardware into three functional units: sensing, communication, and processing. The sensor unit contains two primary components, sensors and digital convertors. Sensors measure a physical quantity such as speed, pressure, temperature, etc. and typically output an analog signal. This signal is in turn converted by the analog to digital convertor (ADC) before being passed to the processing unit. Inside the processing unit an embedded processor handles any desired operations on the digitized sensor data before it is stored or transmitted. However, as internal memory is typically limited on an embedded processor, most systems augment it with external memory (RAM). Additionally most embedded sensor nodes also contain a non-volatile memory (flash) used to store their operating program as well as configuration information. Together the processor, RAM, and flash make up the processing unit of the embedded sensor node. Periodically the processed sensor data is transmitted to remote monitor through the communication unit. While our example cites a transceiver and antenna for wireless communication other methods are often used such as local area networks (LAN). This holds true for the other functional units as well, whose device specific components may vary but general functionality remains static.

![Figure 2.1: The typical hardware configuration for an embedded sensor node. Dashed regions represent functional units of the system and shaded components are those which SWIP is present on.](image-url)

Based on Figure 2.1 we can observe what makes our example embedded system a sensor
node is the functionality of the communication and sensor units. While these units are specific to embedded sensor nodes the processing unit exists on nearly every embedded system. In addition Figure 2.1 also highlights the components that typically would be exposed to the SWIP on such a system. These components almost exclusively reside in the processing unit. The only exception, denoted by the shading of the communications unit, is in the case of a remote software upgrade in which SWIP is transmitted to the device. Figure 2.2 depicts the architecture of an embedded legacy system derived from our observations of the embedded sensor node. The model focuses on only the most common components of an embedded system which are exposed to the SWIP. As such we extract the entirety of the processing unit and simplify the communication unit to a single component which could be wireless, LAN, or even peripheral (USB) based. The nature of this communication being dependent on the context in which is being employed. The architecture model presented in Figure 2.2 will serve as the basis for our analysis of legacy embedded systems.

![Figure 2.2: The architecture associated with the core components of a legacy embedded system that are exposed to SWIP.](image)

### 2.3 System Components

The development and deployment of the targeted legacy system architecture introduced in Figure 2.2 may involve the use of dozens if not hundreds of individual software and hardware components. To simplify our evaluation we define the following broadly grouped system components:

- **Hardware** - The physical silicon and system (PCB, enclosure, etc) on which the SWIP is deployed for storage and processing. A systems hardware is comprised of several other components:

  - **Embedded Processor** - The hardware component responsible for execution and partial storage of the SWIP. This may range from a simple microcontroller...
(PIC) to a more robust digital signal processor (DSP) or even a full fledge processor such as a PowerPC (PPC). Such devices tasked for embedded applications often contain large amounts of high speed internal RAM, power saving features, and specialized components for data processing, peripheral support, etc.

* Program Flash/Rom - Most embedded systems are designed to handle intermittent power supplies such as when a battery is replaced or a device is turned off to conserve power. As such these systems require a non-volatile storage mechanism to maintain their software and configuration between power cycles. On most modern systems this storage is provided through the use of a Flash or EEPROM component.

* External Ram - A system's software could execute in-place on the Flash device on which it is stored. These devices tend to be relatively slow and as a result the contents of flash are typically copied into faster RAM for execution. However with the increasingly complex nature of embedded software, internal RAM does not usually provide adequate storage. This is resolved through the addition of external RAM components.

* Miscellaneous Components - A typically embedded system may have anywhere from dozens to hundreds of additional components. These components provide a wide array of features from power rectification, bus management, signal filtering, and much more. However, while they are often controlled by or act as a conduit for SWIP they typically do not interpret, store, or execute it.

- Software - A complex program that is a collection of code, often from several sources, and compiled into a binary. This binary is executed on the embedded system’s processor and stored in its internal RAM, external RAM, and/or Flash. For the purpose of this work we divide software into two types:

  * Software Intellectual Property (SWIP) - Software which operates on the embedded system and provides a set of unique features of functionality that represents a financial or market advantage to its developer. The nature of this software is such that a developer will inherently seek to protect it from illicit duplication, reverse engineering, or modification. This is the system component that this work seeks to protect.

  * Non Software Intellectual Property (Non-SWIP) - The remaining software which is deployed on an embedded system but is not considered proprietary or critical in nature. This can be simple communication protocols or any other software that is developed by the software developer.

  * Third party Software - Commonly manufacturers of complex components such as processor provide supporting software in the form of chip support libraries or modified versions of standard C/C++ headers. Software developers will often utilize such supporting software to reduce development times.
2.4 System Interests and Trust Boundaries

The development of an embedded system often involves many parties often with conflicting interests. For the purpose of this work we define these involved parties and the trust model between in this section.

2.4.1 Involved Parties

It is very rare that an embedded system is the result of a single developer’s effort. There can be possibly dozens of developers involved that are responsible for different hardware and software components of the system. For the purpose of this work we broadly group these entities into the following groups:

- **Hardware Manufacturer(s)** - The manufacturer(s) of the embedded systems physical components such as the processor, flash, external memory, printed circuit board, physical enclosure, etc. In most development scenarios a single embedded system will have many hardware manufacturers.

- **System Developer** - The developer of the embedded system on which the SWIP will be deployed whose role is often that of hardware integration. In certain instances the system developer may also manufacturer some of the hardware for the system such as the PCB. More complex systems may have more than a single developer.

- **SWIP Developer** - The developer of the application SWIP which operates on the embedded system provided by the system developer. It is often the case that the SWIP developer is the same as the system developer.

- **End User** - The owning individual or entity who will utilize the embedded system either as a standalone product or part of a larger integrated system.

- **Adversary** - A malicious party whose primary goal is the theft of the legacy embedded systems intellectual property. We provide further detail into the adversary’s capabilities during our trust model definition in Section 2.4.2.

2.4.2 Trust Model

The primary concern of the hardware manufacturer is the sale of hardware. As a result, it is in the manufacturer’s interest to provide a secure platform that is more appealing to system or SWIP developers. Likewise the goal of the system developer is to produce a robust full featured platform to attract the end user. To this end, it is in the system developer’s interest to provide a secure, tamper resistant, platform for SWIP developers. The goal of the SWIP developer is to maximize revenue from the sale of its SWIP. Thus, the system that can best provide security against counterfeiting or reverse engineering will attract the
best end user applications and generate better sales. Both the hardware manufacturer and the system developer benefit from increased sales and have an incentive to trust each other. Beyond financial incentives the interleaved nature of an embedded system requires that the hardware and SWIP are considered trusted. Figure 2.3 illustrates these overlapping trust boundaries by the dashed regions.

Often an end-user is only interested in obtaining the best bargain which can include counterfeit systems or pirated SWIP. Worse still is the case when the end-user is malicious, actively seeking to compromise the design for the purposes of reverse engineering or counterfeiting. We base our proposed solution on this adversarial model which is represented in the exclusion of the end user in Figure 2.3. This includes any software that is not directly bound or verified by the design flow (i.e. belong to the trusted parties) and may possibly be malicious. Further we define our adversary as one who has the following properties:

- Are one or more individuals, possibly an industrial competitor.
- Can examine a legacy embedded system in the field.
- Can acquire one or more legacy embedded systems through legitimate or illegitimate means for analysis in private setting.
- Possess comparable technical knowledge to that of the system and software developers.
- Possess comparable technical equipment to that of the system and software developers.
- Wishes to extract the SWIP for analysis, reverse engineering, or duplication.
- Does not have access to the secure development environment.
- Does have access to the user environment and physical hardware.

2.5 Development Design Flow

The creation of an embedded system from concept through hardware and software development to the deployment by an end user is a complex process. As we have shown it involves the combination of numerous components by a myriad of involved parties. Figure 2.4 depicts the general flow of this process in a very rigid way. In a more realistic model there would be significant overlap between the system and software developers. However, as the target of our work is a legacy system, which we define as having a primarily fixed hardware design, we can employ a more rigid design flow. This allows us to focus our efforts on the software development, deployment, and system use as shown by the shaded regions in Figure 2.4.
Figure 2.3: Trust boundary and relationships between system components and development parties. Dashed regions which overlap signify trusted relationship between parties.

Figure 2.4: System design flow from hardware development through end use, shaded regions highlight primary focus of work.
2.5.1 Software Development

We begin exploring the software development design flow, shown in Figure 2.5, with a fixed hardware embedded system design provided by the system developer. Typically a software developer will produce what we term a mixed software distribution, containing both SWIP as well as Non-SWIP. As noted in our system component definition the developer inherently wishes to protect the sensitive SWIP and no vested interest in protecting the Non-SWIP. The source code for the SWIP and Non-SWIP are compiled and linked with chip support or other libraries that are provided by the system developer or hardware manufacturer. These libraries are often proprietary in nature and implement basic functionality of the device such as initialization and peripheral input/output (I/O). The result of the linker operation is a static software binary executable (herein referred to as binary) that contains a mixture of SWIP, Non-SWIP, and third party software. While many binary file formats exist, the most prevalent are the Common Object File Format (COFF) and the Executable and Linkable Format (ELF). However as COFF is slowly being phased out in favor of the more robust ELF our work will focus on the ELF format. Finally the binary is copied into the non-volatile Flash memory on the embedded system by a loading utility provided by the system developer. The hardware embedded system and the programmed binary constitute the final product that will be distributed to the end user.

2.5.2 Software Deployment

The ELF binary format was originally intended to be utilized on full featured processors and operating systems such as Linux. As a result it contains additional information about how an executable should be placed into system memory, symbol tables, and relocation. Most embedded systems however do not have the necessary support, such as dynamic linkers or application binary interfaces (ABI), to handle loading binaries or resolving dynamic dependencies. Rather a software application, provided by the hardware manufacturer or system developer, handles parsing and loading the ELF binary onto the embedded system so it can be executed in place. Figure 2.6 depicts the functionality of such a loading application, herein referred to as the system programmer. During the linking stage of the software development phase a user specified memory map instructs the linker where each software section of the binary will reside in the embedded systems memory. As we depict in Figure 2.6 this information is stored in a small header at the beginning of each section. Contained within the header is the sections name, size, offset inside of the file, its memory mapped location, along with other details. The system programmer parses the binary and extracts the headers along with their corresponding sections. The sections are then copied either directly into the embedded systems non-volatile memory through a programming cable or into an intermediate file format (flash image) to be programmed at a later time. As the system programmer copies each section it discards the additional information stored by the ELF binary ensuring that the deployed binary is of a minimal size. Figure 2.6 (A) depicts the final flash arrangement where each section is positioned for in place execution.
Figure 2.5: System design flow from end of system development through the creation of the final embedded system product.
Figure 2.6: System design flow from end of system development through the creation of the final embedded system product.
A key drawback to completely stripping the ELF information during programming is that the execution is limited to traditionally slow flash. Alternatively some information such as the section size and memory mapped location can be retained during program. At boot a simple boot-loader program can execute in place and much like the system programmer copy each section into its appropriate location in faster RAM. Performing this process during a systems boot is referred to bootstrapping and is a common practice on larger and more complex embedded system designs. Figure 2.6 (B) illustrates the memory map of such a bootstrapped deployment.

2.5.3 Software Execution

The final phase of the legacy embedded system design flow is the operation of the system under the end user. Regardless of the system configuration, execute in place or bootstrapped, at boot the embedded processor begins execution from the binary stored in flash memory. In either case portions of the CSL may be run to initialize additional system components before execution of the main program begins. However, in the bootstrapped scenario the boot-loader copies the contents of each section into internal or external system RAM as described by their boot header. After the boot loader completes initializing the system memory space it transfers execution to the system software. At this point the system is ready to meet the needs of its end user.

2.6 Security Evaluation

Establishing a detailed system model of a legacy embedded system, its development, and operation allows us to better understand the threats that an adversary poses. Referring back to our example system we construct the following usage scenario. A legitimate end user has deployed a significant number of embedded sensor nodes along a highway corridor to automatically monitor road conditions and detect accidents. Left unguarded these sensors rely on their enclosure to protect them from being stolen or tampered with. As the devices age and the needs of the user change, some of the systems may be sold to third parties, recycled, or disposed of. These embedded systems contain proprietary SWIP that is valuable to its developer as it allows the nodes to detect accidents with a high degree of certainty and few false positives. The system and software developers continue to construct new systems while many of the core algorithms, which represent a significant market advantage, remain unchanged. With the potential threat of competitors duplicating their systems capabilities the developers wish to evaluate the threats to their SWIP and possible solutions to providing security retroactively to their deployed system base. With this framing for our evaluation we derive various attacks on the system. This effort is not intended to be all encompassing but to serve as general example of what a capable, resourceful, and intelligent adversary might do. From the list of attacks we can distill a list of critical vulnerabilities and act as a guide for our development choices.
2.6.1 Attack Scenarios

An adversary either stumbles upon or locates a sensor in the field. Curious to its inner workings they use available tools commonly found in a tool chest to either force or gently open the sensor to gain access to its components. As it is the first time that such a system is found they are unable to gain access without triggering a tamper mechanism that erases the SWIP from the device. However once disabled the systems hardware can be disassembled and reverse engineered. Such a complex task would most likely be done in a traditional laboratory setting and yield a better understanding of security measure and how to defeat them. It isn’t long before the adversary is able to locate additional systems, defeat any physical tamper mechanisms, and can open the device successfully allowing them full access to the hardware with SWIP deployed on it. At this phase an attacker can proceed and attempt to recover the SWIP through either passive or active means. A passive attack is one in which the adversary does not alter the functionality of a system. Instead they observe communication to the device, data passing on ports between components, or even the power consumption of the system. Passive are often considered preferable because they are less likely to be detected and may require less technical expertise. The following scenarios are considered to be passive in nature:

- **Copying Flash Memory:**
  - An adversary has gained access to the sensor and removes the flash component of the system which holds the SWIP. Such an approach even be taken with a no longer functioning system whose Flash was never erased before its disposal or resale. The adversary can utilize standard computing solutions such as a laptop or desktop to try to copy and analyze the content of the flash which contains the SWIP. This is the most direct method of stealing the SWIP contained on the system.

- **Internal Data Interception:**
  - The communication links between components of the system may be probed to either intercept data passing between them. Such an approach may yield detailed information into the workings of the system or to copy the SWIP. While it is also considered possible to probe the internal buses of individual hardware components this is considered technically far more challenging.

- **External Data Interception:**
  - The adversary taps into the communications method of the device and intercepts information being transferred to the system. This information may contain sensitive information that could be used to extract the SWIP. Such an attack could be performed with a laptop and commonly available radio equipment. Alternatively an active attack is one in which the adversary exploits a feature or weakness of the system to actively change its operation. By doing so the attack hopes to either expose the SWIP or gain control of the system so
that it can be extracted. Generally speaking active attacks are more difficult technically achieve as they require more knowledge of the system. The tradeoff is that such attacks are more powerful and difficult to protecting against than passive attacks

- **Software Debugging:**
  
  * Many embedded systems contain a software debugging interface that can be used to troubleshoot problems in the field. Once a system has been opened a connection is made to this interface through means such as RS232 and the system debugging capabilities are exploited to compromise or extract information from the system.

- **Buffer Overflow:**
  
  * An adversary with sufficient knowledge of the system and its peripherals designs malicious input from such peripherals to generate a buffer overflow which may be used to gain control of the system. This sort of attack would require specific knowledge of the code base ahead of time or sufficient time in the field to analyze the code.

- **Fault Based Attacks:**
  
  * Following reverse engineering the power, timing, or interrupts may be purposely used incorrectly in order to cause the system to malfunction. Such states could be used to gain access in otherwise unperceived methods. Although this is a trial-and-error process, it is especially dangerous during crucial phases of operation, such as startup.

- **Hardware Debugging:**
  
  * A hardware debugging interface such as JTAG [1] is included on most modern integrated circuits including processors. Once a device is open an attacker can connect to such a debugging port and gain direct access to state of the system hardware. Original attempts will utilize generic boundary scan techniques. With subsequent systems attacked these techniques may become more advanced, since a better understanding of the system will pose a larger threat. Ample time to analyze the sensor and its components is given such that specific JTAG implementations may be used that combat timing or physical protection means. Monitoring systems for JTAG might be disabled or circumvented.

- **Code Injection:**
  
  * An adversary gains access to the device and uses their own memory or interjects with communication of the board to inject malicious code onto the processor. This sort of attack would require specific knowledge of the code base ahead of time or sufficient time in the field to analyze the code.
2.6.2 System Vulnerabilities

The ultimate goal of each of the attacks described in the previous scenarios is to comprise the embedded system so that the SWIP can be removed. While the attacks utilize various vectors and techniques to achieve this access we extrapolate a key set of vulnerabilities that allow these attacks to occur. First and foremost of these is that availability of the SWIP in system components. Generally processor memory and RAM are not as large of an issue as when the system loses power or tamper detection they can be rapidly cleared. However, even in systems with sophisticated anti-tamper mechanisms if the device loses power or is damaged before the flash can be cleared then an attacker can still extract the SWIP contained within due to its non-volatile nature. It should be the goal of the software developer to ensure that code stored in a non-volatile manner is not readily duplicated. A secondary concern is the ability of an attacker to observe the data transfer between internal components as well as external communication of the sensor node. This can be particularly dangerous in the case of the communication between the Flash and embedded processor during the initial boot stage. For the purpose of this work we assume that the attacker can’t go beyond this stage and analyze directly the internal workings of the hardware components. This scenario and that of hardware debugging, code injection, and hardware impersonation all stem from attackers physical access to various busses and interconnects on system. It should be the goal of the system developer and software developer to mitigate the impact of this access. Finally there are the implementation related vulnerabilities such as software debugging, buffer overflow, and fault injection. Most of these issues can be addressed by taking preventative steps during development. For instance a developer can remove software debugging from a system or require a password for its access. Buffer overflow and fault injects could be mitigated through better coding or system development practices. As these vulnerabilities primarily rely on implementation details they are outside the scope of this work.

2.7 Related Work

2.7.1 Software Protection Techniques

The goal of this work is to protect Software Intellectual property by limiting its execution to a particular authorized system [6]. The main goal of this restriction is to prevent the unauthorized duplication or reverse engineering of software. Numerous protection schemes have been proposed to address this issue including watermarking, tamper-resistance, obfuscation [18], and encryption. Tamper resistance allows a program to validate its own integrity.
and to cease operation if it has been modified [30, 16, 34]. Water-marking incorporates a developer signature into a program to detect intellectual property theft and reuse [71, 57]. Obfuscation transforms a program in such a way that it is hard to reconstruct the source or assembly code from a static program image [47, 25]. Much like obfuscation, encryption transforms a program in such a way that it is rendered unreadable. However, in encryption this is performed by a cryptographic algorithm and the result is no longer executable. Decryption is first required and can only be performed by those possessing special knowledge such as a decryption key. Of these techniques encryption can provide a provable level of security, a minimum of overhead, and has well understood and simple to implement public algorithms. As such its application is very desirable for securing SWIP and there have been many applications proposed for applying them to embedded system. However, when any of these approaches are deployed purely in a software context, they are vulnerable to virtualization techniques [6] and exploitable features in modern processors such as virtual memory support [67]. To address this issue, a protected execution environment is necessary for many of these techniques.

### 2.7.2 Existing System Solutions

Any system attempting to provide a protected environment for storage and execution of software must operate as a trusted device. This concept of trusted computing is implemented by the Trusted Computing Group’s (TCG) Trusted Platform Module (TPM) [27]. A TPM functions by providing a secure hardware medium for storage and generation of keys or certificates, mechanisms for monitoring a processor, and identifying a system. This concept of a secure platform is further expanded by AEGIS architecture proposed by Suh et al. [64]. AEGIS provides for a single-chip processor with built-in tamper resistance and detection as well as dedicated cryptographic components. Alternatively, Lee et al. proposes a Secret Protected (SP) architecture [41] which unlike AEGIS or TPM represents a paradigm shift, where data is tied to a user rather than a particular device. The SP architecture is an attractive solution as its operation is primarily software driven by a Trusted Software Module (TSM). However, to ensure the TSM’s integrity a processor must support a Concealed Execution Mode (CEM) that isolates TSM functionality and that contains volatile cryptographic keys. The hardware supported CEM allows the TSM to perform a trusted bootstrap, as proposed in [27, 5, 60], and verify the TSM binary and system state. Such a hardware assisted bootstrap mechanism is considered critical to establish a trusted execution platform that is robust against a variety of attacks. Lie et al [42] provides an alternative trusted execution platform through the concept of execute only memory (XOM). The XOM architecture ensures certain private portions of memory once written are only executable and can’t be modified or read. However, like AEGIS and SP, XOM requires a specialized hardware implementation and a shift from traditional architectures.
2.8 Summary

In this chapter we have:

- Introduced the concept of a legacy embedded system.
- Defined the components of a legacy embedded system by examining an embedded sensor node.
- Provided a detail overview of the software design flow for a legacy embedded system.
- Defined the parties involved in the development of a system and their trust boundaries.
- Evaluated security concerns in respected to a legacy embedded system.
- Presented related work to defending embedded systems and noted their legacy applications.
- Identified software encryption as a critical first step to securing SWIP.
Chapter 3

Securing SWIP on a Legacy Embedded Systems

Evaluation of our example legacy embedded system highlights that the largest vulnerability is the storage of its SWIP in non-volatile memory. If the system is disabled, destroyed, or compromised an attacker could easily copy these contents. During the course of our literature overview we explored possible solutions of which SWIP encryption was the best suited due to the simplicity of implementation, its level of security, and minimum overhead. Existing SWIP encryption solutions lend themselves to hardware based approaches for securing newly developed systems [27]. However, this leaves a great deal of older systems that run on a legacy hardware vulnerable. Rather than opting for costly hardware retrofits for such systems, a software approach may extend the platform’s useful application life. This section of our work presents such a software-only solution for securing Software Intellectual Property on a legacy embedded system.

3.1 Overview

Figure 3.1 illustrates our three part solution and its integration into the software development design flow. First, tight integration of SWIP encryption and the software tool-chain provides a novel and streamlined method for the protection of SWIP. Second, system software is supplemented with the addition of a Security Kernel (SK) which provides a platform for the decryption of encrypted code at boot. Finally, the security kernel takes advantage of the embedded sensors communication link to negotiate a SWIP decryption key from a key server at startup. The use of a key server is required as our example legacy system does not possess any secure nonvolatile memory. Under the generic nature of the implementation it cannot be assumed there is hardware present that does.

Despite the end application being a unified system, the separation of system components serves two purposes. First and foremost the application of a complete cryptographic solution to a difficult problem, such as intellectual property protection, is very technically difficult. There are dozens of aspects such as key management, communication protocols, system integration, etc. that must be considered. By providing standalone components this procedure can be deployed incrementally allowing for greater value to be returned in a shorter time period. Second and most importantly, cryptographic functions and protocols are very complex. That is not to say that their implementation is necessarily difficult to perform as
there are many excellent reference implementations for comparison. Any modifications to these protocols may inadvertently introduce weakness into the cryptographic element. The cryptographic elements employed like Advanced Encryption Standard (AES) [52] are widely deployed and have had many years of development and scrutiny to provide the highest level of security possible. As such it is only natural to try to isolate the cryptographic elements into standalone packages in order to limit their modification. This in no way limits the application of alternative implementations of these components as long as their functionality can be properly verified. To ensure compatibility we ensure that all additions to the tool-chain are also designed to be implemented in portable C code. Additionally we strive to maintain a minimum of 128bit AES secret key security or equivalent [26] as specified by the NSA guidelines [56] where possible. The following sections provide an overview of each of the major components of the IPSEC system. An ideal implementation is outlined for the purposes of this report, however in practice some compromises will be required to maintain performance and security requirements.

3.2 Software Intellectual Property Encryption

Software intellectual property encryption represents the core of our protection scheme and is broken down into two steps: identification and encryption. In the identification phase the SWIP code sections that require protection are marked for encryption. In the encryption phase, code sections are encrypted with a generated symmetric key. The encrypted sections are then reinserted into the binary for deployment and eventual decryption on the legacy embedded system.

3.2.1 Identification

Ideally, all software would be encrypted to prevent any reverse engineering attempts. However, a system cannot boot from a completely encrypted executable. Therefore, there must be a mechanism which will first retrieve the SWIP decryption key from the remote server and decrypt the SWIP before execution can begin. Though specialized hardware
may be employed to perform this task, such a system could not be feasibly deployed on a legacy embedded system. Instead, a software based Security Kernel is introduced that remains unencrypted. The result is a system that contains two major types of software. One consists of the system software components defined in Section 2.3 including SWIP, Non-SWIP, and third party software, which are all encrypted. The other is the SK and its supporting software which remains in plain text to function as a secure boot-loader for the embedded system. Identification of sections that will remain unencrypted is accomplished during development through the use of built in section attribute made available by most C/C++ compilers. We give an example of the use of such an attribute for the 4.2.0 GNU Compiler Collection to place a variable in a section entitled foo.data and a function in a section entitled foo.code. The implementation of such features may vary between platforms but similar facilities are typically available.

```c
static int var[100] __attribute__((section ("foo.data"));
void foo( int *var ) __attribute__((section ("foo.code"));
```

Sections specific to the SK are identified in the same manner but separated from supporting code to facilitate addition processing and modification applied during the encryption process. This is done through the use of a naming convention where all section names preceded by secKern are marked as belonging to the SK. The inclusions of the SK and section identification are the only two aspects of our solution that involve the direct modification to the system software design.

### 3.2.2 Encryption

Encryption of the specified software sections occurs after the compilation and linking of the design results in a complete binary and is a post processing step. As we have adopted the strategy of allowing individual sections of software to be encrypted it is necessary that these sections are logical entities handled by the compiler and linker. As such we obtain tight integration between software encryption and development. Figure 3.2 shows an unprotected binary containing SWIP and our SK in the ELF format as explored in Section 2.5.2. Rather than attempting to directly parse the ELF binary we take advantage of a development tool included with most compilers, objdump, to output detailed information about binary files. By using objdump we can easily output a list of all the sections in a binary and retrieve their headers which contain information about the section name, file offset, and size. Based on the C attribute functionality and our naming convention we can easily identity sections that require encryption. Passing this information as a flat file to our Intellectual Property Encryptor (IPE) utility it can easily retrieve each software section from the unprotected binary by using its offset and size. As each section is extracted the IPE encrypts it using AES, the application of which is further discussed in Section 3.2.4, before writing the encrypted section back into the binary. For additional security the IPE allows the use of different keys and nonce to be used on different code section. This provides a greater flexibility in
key and IP management by allowing the developer to specify different key distribution and management policies for each section.

Besides the encryption of software sections, the IPE also creates an additional data section in the resulting encrypted binary. Space for this data section is allotted in the security kernel and denoted by a special section name. The plain text data section holds the offset, size, and decryption information for each software section that was encrypted. This data section is used by the Security Kernel at boot to locate encrypted software for decryption and deployment to its proper location. After the IPE concludes the resulting binary will contain both encrypted and plain text code sections. Any standard programming utility may be used to deploy the protected binary to its target system. Encryption of the Elf binary by the IPE and its deployment all occur within the trusted development environment and are not subject to attacks.

### 3.2.3 Decryption

After a protected binary is deployed onto an embedded system it is the responsibility of the SK to retrieve the SWIP key from the server and decrypt the SWIP. In this way the SK functions much like the IPE except in reverse as shown in Figure 3.3. As the system boots the SK uses the headers and decryption information stored in it by the IPE to locate and decrypt the protected software sections. At this stage the SK takes on the role of a bootloader and copies the decrypted sections into the memory mapped located in the processor memory denoted by their headers. After every section is decrypted execution is transferred over to the processor memory and proceeds as normal.

### 3.2.4 Advance Encryption Standard

The Advanced Encryption Standard (AES) [52] is a natural selection for application in the IPE. Not only is the block cipher well known and but there exist many freely available implementations that stress performance or code size, yielding a great deal of design flexibility. When used with a key strength of 128-bits AES also meets the requirements for secret level clearance as specified by NSA standards [56]. A detailed evaluation of AES is outside the scope of this work. However, in Figure 3.4 we provide an overview of the general functionality of a block cipher. Give a key, the block cipher takes a fixed size plain text (unencrypted) input called a block and transforms (encryption) it into a cipher text (encrypted). To obtain the plaintext from the cipher text the reverse transformation (de- cription) is performed. The nature of the decryption transformation is such that it is very difficult to perform without the key. As AES is a symmetric key cipher the same key is used to both encryption and decryption. This basic operation of a block cipher such as AES is referred to as Electronic Codebook (ECB) [53] mode and has very significant drawbacks. Primarily that identical input blocks encrypted with the same key will result in the same cipher text which can leak information about the system.
Figure 3.2: Encryption of an Elf binary by the IPE.

Figure 3.3: Decryption of software sections by the SK at boot.
Figure 3.4: Block cipher, such as AES, basic mode of operation known as ECB.
Due to the inherent issues with ECB, encryption in the IPE is performed through the use of AES in Counter Mode [46] as shown in Figure 3.5. In counter mode a unique counter is generated for each message block and encrypted. The resulting output (key stream) is then 'xored' with the plain text to generate the cipher text. To minimize the chance that counters are reused a nonce, or unique number, is used for every software section decrypted. As such the counter mode of operation provides several advantages over ECB. First, only the encryption must be implemented as both encryption and decryption generate their key stream through AES encryption. Second, the key stream allows for the encryption of arbitrarily sized messages so code sections do not have to be a multiple of the AES block size (128-bits).

![Figure 3.5: Counter mode of operation for a block cipher.](image)

3.3 SWIP Key Transmission

Decryption of the protected software, stored in the system non-volatile memory, at boot requires the presence of the 128-bit AES key. The simplest method of storing the key would be to place in non-volatile memory alongside the encrypted SWIP. However if an attacker copied the flash contents they would only have to analyze the SK to determine the location of the key and encryption algorithm used. With this information it would be a trivial task for the attacker to decrypt and steal the protected SWIP contained in the system. We solve the issue of key storage by taking advantage of the communications capabilities of the example embedded system to store the SWIP on a remote server. At boot before the SK begins decryption it negotiates with the remote key server to obtain the decryption key and stores a local copy on the processor memory. Once decryption is completed the key is destroyed such that it never exists in a non-volatile form on the system. However we cannot assume that the network used to perform this exchange is secure. Rather we must view it as an open channel where it is relatively trivial for someone to observe or transmit information. As a result it is unsuitable for relaying sensitive system information such as the SWIP key. In this section we identify the requirements needed for a secure communication link and examine methods for establishing such a link within the context of a legacy embedded system.
3.3.1 Secure Channel Requirements

An open channel is considered a method of communication which allows for the arbitrary reception and transmission of information by any party. Alternatively, a secure channel is one that is resistant to reception or transmission by unauthorized parties. Such a secure channel relies on either a technical or computational challenges to achieve these properties. For the purposes of example system we assume the establishment of a point-to-point secure channel between the sensor and a central server. In this arrangement the only parties that are able to communicate or observe the data transmitted over channel are the server and a single embedded system. A point-to-point secure channel can physically span several other devices which may act as relays to transmit the channel. However, due to the features of the secure channel these devices cannot observe the data passing inside the channel. This results in a communication model that is logically equivalent to a direct connection between the server and embedded system. Establishing a secure channel requires that two principles are satisfied:

- **Confidentiality**: Information is accessible only to those authorized to have access, such as the parties involved in the communication.

- **Authenticity**: Communication over the channel comes from a genuine known source that can be validated.

Confidentiality is achieved by leveraging AES based encryption and an elliptic curve based public key exchange. Authentication however proves to be a much more difficult challenge is discussed in further detail in Sections 4 of this work.

3.3.2 Confidentiality

The primary goal of establishing a secure channel is to facilitate the transmission of the SWIP key. However, as our example system utilizes an open communication standard it does not provide a guarantee of confidentiality for the transmission. Instead, we overcome this by encrypting data before it is transmitted through the use of the AES as discussed in Section 3.2.4. This allows for the creation of a confidential channel between two points, the embedded sensor and a remote server. As with our implementation of the IPE we employee a 128-bit AES key to guarantee a minimum of security deemed appropriate for secret level information. Relying solely on AES for creating a confidential channel creates a critical problem. As a symmetric key block cipher it requires the same key for encryption and decryption. This would result in a key management challenge as each system would have to be programmed with a unique key then a copy stored on the server. As systems moved between multiple servers such an issue could quickly become intractable. More importantly however is the storage of such a key would have to be done in non-volatile system memory and face the same risks as directly storing the SWIP key. We address the management and storage of communication keys through the use of an Elliptic Curve Diffie-Hellman (ECDH)
key exchange protocol [55]. ECDH relies on an Elliptic Curve (EC) [32] point multiplication to establish an identical key across an open channel. Such an exchange is illustrated in Figure 3.6 and starts with the embedded system requesting a secure channel to the key server. The server and sensor generate a random integer which is combined with a common base point and used to derive a public key. The public key is transmitted across the open channel and used to derive an identical private key on both the server and sensor. However, due to the properties of EC point multiplication, it is mathematically difficult to determine the random integers or final value based on the transmitted public keys or base point. A more detailed evaluation of ECDH’s functionality is provided in Section 3.3.3.

![Figure 3.6: Establishing a confidential channel utilizing ECDH key exchange and AES.](image)

The private key generated by the ECDH exchange is 512-bits long while AES requires a 128-bit key. Reduction of the shared secret to the appropriate key length is performed by the application of the Davies-Meyer hash. The Davies-Meyer algorithm achieves this by using the same AES encryption function used for transmitting the SWIP key to create a cryptographic hash function. The reuse of AES for software encryption, confidential communication, and hashing helps minimize the overhead for SK implementation. Having derived an identical symmetric key for AES the key server can now encrypt the SWIP key and transmit it across the confidential channel to the embedded system. There the SWIP can be decrypted and passed onto the SK so that the system software can be decrypted and the system can begin execution. Combining the Elliptic Curve Diffie-Hellman and AES cryptographic primitives we can establish a confidential channel by exchanging three messages in total as summarized in Table 3.1.
Table 3.1: Messages transmitted during ECDH Key Exchange and distribution of SWIP key.

<table>
<thead>
<tr>
<th>Message</th>
<th>Size (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensor Public Key</td>
<td>64</td>
<td>Embedded system public key derived from P256 base point and randomly generated integer.</td>
</tr>
<tr>
<td>Server Public Key</td>
<td>64</td>
<td>Key server public key derived from P256 base point and randomly generated integer.</td>
</tr>
<tr>
<td>Encrypted IPE Key</td>
<td>16</td>
<td>IPE Key encrypted with key derived from the private key resulting from the ECDH exchange.</td>
</tr>
</tbody>
</table>

3.3.3 Elliptic Curve Diffie-Hellman Key Exchange

Elliptic Curve arithmetic is built on top of modular arithmetic, and creates public and secret keys by multiplying a point on an elliptic curve by a scalar value. By default, points are represent in the affine \((X,Y)\) coordinate system. For efficiency reasons, embedded system implementations internally apply projective format \((X,Y,Z)\) for points, and include conversions from/to affine to projective format as needed. The IEEE standard [32] provides generic implementations for addition, subtraction, doubling, conversion between affine and projective, and scalar multiplication. The modular arithmetic for point operations are based on finite field arithmetic of either the prime \(GF(p)\) or binary \(GF(2^P)\) type. Our design uses the \(GF(p)\) scheme, in part because an easily accessible open-source implementation are available that could be used development references. Field length for the finite arithmetic is another system parameter to choose. According to the NSA guidelines [56, 12, 51], the equivalent to 128-bit AES security necessary for secret level information is a minimum of 256-bits under prime field ECC. Figure 3.7 illustrates in detail the ECDH key exchange protocol where the following is a list of the variables employed in the algorithm.

- \(G\) - NIST P-256 prime field elliptic curve base point for ECDH.
  - * 512 elliptic curve point
- \(n\) - Random 256 bit integer generated by the server to multiply against the base point and generate its public key from. This value needs to be kept secret.
  - * 256 bit integer
- \(z\) - Random 256 bit integer generated by the sensor to multiply against the base point and generate its public key from. This value needs to be kept secret.
  - * 256 bit integer
- $P_D$ - Key server public key generated by multiplying $G$ by $n$.
  * 512 elliptic curve point
- $P_S$ - Embedded system public key generated by multiplying $G$ by $z$.
  * 512 elliptic curve point
- $K$ - Resulting private key derived from multiplying other party’s public key by its own random integer. This value is used as a key for the AES channel once reduced by the Davies-Mayer Hash.
  * 512 elliptic curve point

![Diagram of Elliptic Curve Diffie-Hellman public key exchange.](Diagram.png)

Figure 3.7: Elliptic Curve Diffie-Hellman public key exchange.

Based on the defined variables the elliptic curve Diffie-Hellman key exchange proceeds as follows:

1. **Initial State:**
   - (a) Embedded system initialized with elliptic curve P-256 base point ($G$).
   - (b) Key server initialized with elliptic curve P-256 base point ($G$).
2. **Embedded system requests a key exchange by generating and transmitting its public key.**
   - (a) A random integer ($z$) is chosen and multiplied against the P-256 base point ($G$) to derive the embedded system public key ($P_S$).
   - (b) Public key ($P_S$) is transmitted to the key server.
3. **Server responds by generating and transmitting its public key.**
(a) A random integer \((n)\) is chosen and multiplied against the P-256 base point \((G)\) to derive the key server public key \((P_D)\).

(b) The embedded system public key \((P_S)\) is multiplied by the server random integer \((n)\) to derive the shared secret \((K)\).

(c) The server transmits its public key \((P_D)\) to the embedded system.

(4) Embedded system generates the shared secret.

(a) The server public key \((P_D)\) is multiplied by the server random integer \((z)\) to derive the shared secret \((K)\).

At this point the AES key can be derived from the shared secret through the use of the Davies-Mayer hash. This entire process can safely be performed over the insecure channel.

### 3.3.4 Davies Mayer Hash

As we employ a NIST recommended prime 256 bit curve that yields a shared secret key that is 512 bits in size. However, the selected key length for AES only requires 128 bits, rather than discard the additional bits which would effectively eliminate some of the entropy provided by ECDH and reduce its security. Instead, a Davies-Mayer Hash (DMH), which uses AES in a feedback arrangement, reduces the private key into 128 bits suitable for use with AES. This distributes the additional 384 bits of entropy provided by the ECDH secret key, evenly across the 128 bit AES key. Figure 3.8 illustrates the construction of the DMH where starting with an arbitrarily chosen initial hash the shared secret is used as a key for AES. The resulting output is then xored with the initial hash input into AES and generates the next hash block. With each round an additional 128-bits of the shared secret are used as a key on the next hash block until all 512 bits are utilized. The resulting 128-bit final hash is then ready to be used as an AES key for the transmission of the SWIP key.

### 3.4 Design Flow Summary

The software components and required modifications to the design flow are summarized in Figure 3.9 from a developer's perspective with shaded elements denoting main components of our design flow. Our proposed system is able to achieve a high flexibility for several reasons. First, we do not rely on any specific hardware components or capabilities outside of those typically associated with an embedded sensor node. Our design only requires the addition of a Security Kernel, the labeling of the SWIP, and the availability of a key server. Second, by maintaining the majority of our functionality in software, we ensure rapid substitution of components, such as the various cryptographic primitives to meet the specific needs of a developer. Finally, by gearing our design flow to use standard C functionality we ensure compatibility across a wide array of system components.
Figure 3.8: Davies Mayer hash construction using a block cipher.
Figure 3.9: Design flow overview, shaded blocks are additions or modifications.
3.5 Security Evaluation

By encrypting SWIP stored on non-volatile flash memory and placing the decryption key on a remote key server we successfully address our major concern over the simple copying of SWIP. While we are able to do this with a software-only solution this still leaves several major issues to be addressed. First and foremost without authentication of our key transmission we run the risk of an attacker impersonating an embedded system to retrieve the SWIP key. Solving this issue is outside the capability of a software only solution and is further discussed in Section 4 with a minimal hardware addition. Second, while protecting SWIP while it’s stored in non-volatile memory is important the attacker still is capable of physically accessing the busses and interconnects of the embedded system. As such a whole host of attacks such as hardware debugging, code injection, and hardware impersonation allow an attacker to subvert our system and extract the SWIP at runtime. This is a limitation of a software-only approach and the nature of a legacy system. However there exists some physical means of protecting from these attacks such as potting, where a device is covered in a resin or other material to prevent physical access to the components. During our exploration of the development of a new FPGA based commodity system in Section 5 we explore alternative solutions enabled by the unique characteristics of FPGA based systems.

3.6 Summary

In this chapter we have:

- Provided an overview of our SWIP encryption solution.
- Gone into detail about how the contents of a binary are identified and encrypted.
- Introduced our Security Kernel that handles SWIP decryption a boot.
- Presented the concept of key storage on a remote key server.
- Defined the concept of a secure communications channel that is confidential and authenticated.
- Introduced Elliptic Curve Diffie-Hellman key exchange for establishing a confidential channel.
- Performed a security evaluation of our partial solution of SWIP encryption and a remote key server.
- Provided background on cryptographic protocols used in our solution including, AES, ECDH, and DMH.
Chapter 4

Authentication

A requirement for establishing a secure communications channel for the transmission of the SWIP key, as set forth in Section 3.3.1, is authenticity. They key server and the embedded system requesting a key form it must be able to mutually authenticate each other. In this section we provide a proof of concept attack against ECDH to illustrate the need for authentication. We then continue on to explore an optional authentication methodology before presenting a modified key exchange protocol to provide authenticity to our secure communication channel.

4.1 ECDH Requires Authentication

The Elliptic Curve Diffie-Hellman key exchange protocol presented in Section 3.3.3 provides a powerful mechanism for establishing a confidential communication channel over an open connection. One of the primary benefits of a public key protocol such as ECDH is that the parties involved do not have to have knowledge of each other ahead of time. No secret values or keys have to be programmed into the embedded system or server before they are distributed. Rather a key can be established between the two, securely, at run time, and over an un-secure channel. However ECDH does not guarantee the identity of those involved in the exchange and leaves the key server to attack.

4.1.1 Simple Attack

The simplest way for an attacker to exploit a lack of authentication is to impersonate the embedded system and request a SWIP key from the key server. Accomplishing this sort of attack is feasible if the enough information about the systems functionality can be gathered through observing communication or obtaining documentation. Once an attacker can reconstruct the exchange (which is based on a well known protocol) they can attempt to trick the key server into issuing them a decryption key. Resolving this issue seems trivial at first glance, the embedded system could simply transmit an authentication code to the key server. While this does eliminate one of the key advantages of ECDH, by introducing a requirement to store a secret for authentication, it does not render it useless. Rather, ECDH
still allows for trivial key exchange such that as a key ages (is used) and its security decreases a new secure channel can be establish.

4.1.2 Man in the Middle

Like all public key schemes, that do not incorporate authentication, ECDH is vulnerable to an attack where an adversary intercepts communications between two parties. This type of attack is commonly referred to as a Man in the Middle Attack due to the attacker’s position in the communication exchange. We illustrate such an attack on a simplified version of our SWIP key exchange protocol in Figure 4.1. The attack begins as the attacker observes the embedded systems requesting a SWIP key from the key server. As the embedded system and key server begin their exchange the attacker positions its self in between the pair such that it can intercept any communications. Next the embedded system, having generated its public key $D$, transmits they key to what it believes is the key server. At the same time the key server generates its public key $S$ and transmits it to what it believes is the embedded system. The attacker having intercepted both messages responds with its own public key $A1$ to the embedded system and the public key $A2$ to the key server. The embedded system and attacker establish private key 1 while the attacker and key server establish private key 2.

Having established a private key with both the embedded system and key server the attacker can now observer communications from one to the other. As the key server transmits the SWIP key to what it thinks is the embedded system it encrypts it with private key 2. The attacker intercepts this message and decrypts it, extracting the SWIP key, before encrypting it with private key 1 and transmitting it to the embedded system. In this way not only does the attacker retrieve the SWIP but they do it in such a way that neither the embedded system nor the key server can detect. In order to provide a robust and secure communication channel a unified protocol is needed.

4.2 Hardware Supported Authentication

Our legacy embedded system model does not possess a secure storage facility for maintaining sensitive information such as the SWIP key. Traditionally such a component would be a secure, tamper resistant, battery backed, volatile memory component that erases a key if the device is tampered with. Incorporating such a component into an existing system would require a complete redesign of the current hardware which is infeasible. For this reason our solution incorporates a remote key server which can provide a robust verification to the authenticity and status of the embedded system before a key is released. However, to avoid impersonation this requires that both the embedded system and key server show knowledge of a secret value. The lack of a secure storage facility is a hard limit of our software-only approach to securing SWIP on a legacy embedded system. In order to move beyond this limitation we propose a compromise in introducing the 1-Kbit Protected 1-WIRE EEPROM with SHA-1 Engine DS2432 from Dallas-Maxim. This simple hardware component can be
Figure 4.1: Example of Man in the Middle Attack on a simplified version of our SWIP key exchange protocol.
retroactively deployed on existing systems without requiring major hardware modifications due to its minimal interface and very low power requirements. We begin by first exploring the functionality of this component before moving on to its application in our system.

4.2.1 DS2432 Overview

The Dallas-Maxim DS2432 component functions similarly to a traditional EEPROM as it provides for the nonvolatile storage of up to 1024 bits of information. However, unlike a typical EEPROM the DS2432 includes a SHA-1 engine that provides an authentication mechanism used to read and write from the device. The 1128 bits of storage that the DS2432 provides is broken down into four pages of 256 bits each, a 64 bit write only secret, five 8 bit general purpose registers, and a 64 bit laser etched identifier. A user interfaces with the device through a 64 bit scratch register where data is stored and verified before a read or writes to the device. Verification is accomplished by the DS2432 requiring a 160 bit Message Authentication Code (MAC) using a Secure Hash Algorithm 1 (SHA-1) engine with the input of a 512 bit combination of the contents of the user memory, laser etched identifier, registers, and secret value. Generating this MAC is the basis for the authentication and indicates knowledge of the secret 64 bit value without exposing the actual value. This is a result of the properties of SHA-1 that make it difficult to reconstruct hashed data from the hash result. SHA-1 is further discussed in Section 4.2.2. An additional feature of the DS2432 is its simple one wire interface for communication. All address, command, and data are transmitted over a single line while the device operates off a parasitic power system and only requires an additional ground connection. As a result the DS2432 is easy to incorporate into an existing hardware system. Figure 4.2 below illustrates the structure of the DS2432.

4.2.2 SHA-1 and Message Authentication Codes

The Secure Hash Algorithm 1 (SHA-1) [54] is a cryptographic hash function which takes an arbitrarily long input and digests it down to a fixed sized output. Cryptographic hash functions differ greatly from traditional hash functions as they possess several defined properties that make them more suited for cryptographic applications such as generating MAC’s. Most important of these is that given a digest it is infeasible to reconstruct the original message and that it is infeasible to find two messages with the same digest. Message Authentication Code’s (MAC) are generated using an algorithm such as SHA-1 by combining a secret key and a public message as inputs to the hash. In this way if two entities are given the same message, they will only be able to produce the appropriate digest if they know the key as well. Likewise given the resulting digest with the message it should not be possible to reconstruct the key. In this way a message can be authenticated securely, even over an open channel. Recently concerns have been raised over the security of the SHA-1 algorithm as collision attacks have been demonstrated where an attacker can find two message inputs that result in the same digest. This however is not of concern for the application of SHA-1 to the generation of MAC’s as its intent is to hide the authentication key while signing the
Figure 4.2: DS2432 overview, write MAC is verified by SHA-1 engine and read MAC is generated by the SHA-1 engine.
message rather than verify the integrity of the message. If an attacker is able to find an input (message + key) capable of generating the same MAC it is unlikely that the key

4.2.3 Writing to the DS2432

Writing to the DS2432 is accomplished first by copying a 64 bit value to the scratch pad. The user must then provide the DS2432 with a 160 bit write authentication MAC using the contents of the scratch pad, the identifier, a data page, the write only secret, and certain registers. At the same time the DS2432 generates this value and compares the user response to its own internally generated value. If they match the contents of the addressed 64 bits in the page are updated with the contents of the scratchpad. Figure 4.3 below illustrates the data registers utilized in this process.

![Figure 4.3: Data utilization to generate DS2432 write MAC.](image)

4.2.4 Authenticated Reading from the DS2432

Performing an authenticated read from the DS2432 is accomplished first by copying a 24 bit value to the scratch pad. The DS2432 will then begin generating a 160 bit read authentication MAC using the 24 bits in the scratch pad, the identifier, a data page, the write only secret, and certain registers. A user with knowledge of the key then can generate this
MAC and verify that the read has come from a genuine device. Figure 4.4 below illustrates the data registers utilized in this process.

![Figure 4.4: Data utilization to generate a DS2432 read MAC.](image)

### 4.2.5 Read and Write Based Authentication

Authentication can be achieved by taking advantage of the DS2432 to perform a two way authentication between the embedded system and key server. This is accomplished by using the properties of reading and writing to the DS2432 as detailed in the previous sections. Creating a valid write MAC for a particular challenge illustrates that the creator has knowledge of the write only secret. This allows the server to authenticate itself to the embedded system node. Likewise generating the read MAC requires that the embedded system have a DS2432 with the write only secret stored inside of it. An attacker would require this value to impersonate either embedded system or node which is difficult to remove from the DS2432 or key server which is stored in a secure location.

### 4.3 DS2432 Based Authentication

Our application of the DS2432 for authentication is illustrated in Figure 4.5. As with confidentiality, authentication is performed between the key server and a single embedded system. The embedded system begins the exchange by requesting authentication from the
A random challenge is then generated and signed by the server using a software model of the embedded systems DS2432 containing the authentication key. Verification of the challenge signature is performed by the embedded system DS2432 and if successful establishes the identity of the server. While the DS2432 is capable of generating a unique response to the server challenge the resulting MAC utilizes 488 out of 512 bits that were used to generate $MAC_C$. Additional security is provided by utilizing the Public Key Derivation Function 1 (PBKDF1) on the response $MAC_R$ as discussed in Section 4.3.1 to generate $MAC_P$.

![Figure 4.5: Authentication overview.](image)

Upon verification of the response the embedded system is proven authentic and the SWIP key can be transferred over the confidential and authentic channel from the server to the embedded system. However, a weakness still exists in the channel as confidentiality and authenticity are not bound together. The messages exchanged for authentication is summarized in Table 4.1.

Authentication using this method requires an interface to the DS2432 on the embedded system and a software simulation of the device on the key server. These two components must be initialized with identical information before the embedded system can be deployed in the field. Great care needs to be taken that the DS2432 chip is not removed from the embedded system without being detected. While an attacker could not retrieve the key they could simply query the device to retrieve a valid response. In this way the DS2432 is primarily targeted at protecting against impostor or clone devices rather than compromised embedded systems.

### 4.3.1 Password-Based Key Derivation Function 1

PBKDF1 functions by performing a software SHA-1 hash on the response $MAC_R$ a random number (up to 1024) of times. An attacker would not only require knowledge of the
Table 4.1: Messages exchanged for DS2432 based authentication.

<table>
<thead>
<tr>
<th>Message</th>
<th>Size (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request</td>
<td>8</td>
<td>Arbitrary request containing who the platform claims to be.</td>
</tr>
<tr>
<td>Challenge</td>
<td>8</td>
<td>Challenge issued by the server to the embedded system to verify its identity.</td>
</tr>
<tr>
<td>$MAC_C$</td>
<td>20</td>
<td>Signature of the challenge generated with knowledge of the secret authentication value proving the identity of the server.</td>
</tr>
<tr>
<td>$MAC_P$</td>
<td>20</td>
<td>Signature response to the challenge generated with knowledge of the secret authentication value proving the identity of the embedded system.</td>
</tr>
</tbody>
</table>
internal 512-bits that the DS2432 utilizes to generate a MAC but additionally they would also have to guess the random integer used in PBKDF1. This increases the computational time required to attempt a brute force attack on the response by a factor of 210. On the other hand the server knows the contents of the embedded systems DS2432 and only requires 210 time total to verify the response.

4.4 DS2432 Authenticated ECDH

DS2432 Authenticated ECDH (DA-ECDH) addresses the issue of man in the middle attacks by incorporating both confidentiality and authentication into a unified protocol. During the course of the public key exchanged used in establishing the confidential channel, the public keys are signed and as a result authenticated as coming from a legitimate source. If an attacker attempts to perform a man in the middle attack they will not be able to generate the signatures for the public key. A detailed overview of the DA-ECDH protocol is presented in the following section.

4.4.1 Overview

The DA-ECDH protocol establishes a secure channel between a server and a embedded system that is both confidential and authentic. Figure 4.6 presents a detailed overview of the DS2432 Authenticated ECDH protocol starting with an ECDH key exchange in dark blue. ECDH is signed by the DS2342 enabled authentication scheme in red before finally establishing a confidential and authentic channel in green.

Details on the messages exchanged during the DA-ECDH protocol are given in Table 4.2 below. In the following sections each of the variables and functions involved in the exchange are defined before a detailed run through of the protocols operation is presented.

4.4.2 Variable Definitions

The following is a list of the variables employed in the DS2432 authenticated ECDH key exchange.

- $K_A$ - Embedded system authentication key stored inside DS2432 hardware on the embedded system and in DS2432 software on the server. This value needs to be kept secret.
  
  - 64 bits

- $D_{A0}$ - Data stored inside DS2432 hardware on the embedded system and in the DS2432 software on the server. This is information other than the authentication key and is considered secret.
Figure 4.6: DS2432 Authenticated ECDH unified authentication and key exchange protocol.

Table 4.2: Authenticated ECDH protocol exchange details.

<table>
<thead>
<tr>
<th>Message</th>
<th>Size (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS</td>
<td>64</td>
<td>Embedded system public key used to establish the confidential channel.</td>
</tr>
<tr>
<td>PD</td>
<td>64</td>
<td>Server public key used to establish the confidential channel.</td>
</tr>
<tr>
<td>MACW</td>
<td>20</td>
<td>Signature of the shared secret used to authenticate the channel and public key exchange. This acts as a server challenge as well as validates server’s identity.</td>
</tr>
<tr>
<td>RSP</td>
<td>20</td>
<td>Encrypted response of the embedded system that validates the embedded system’s identity. Establish the confidential/authentic channel.</td>
</tr>
</tbody>
</table>
* 384 bit user data

- \( D_{A1} \) - Data that will be stored in DS2432 after a successful authentication. A failed authentication cannot modify the internal data of DS2432 more than once.

* 384 bit user data

- \( D_{A, \text{PRevIOUS}} \) - Previous value of DA used to resynchronize in case of failed authentication.

* 384 bit user data

- \( D_{\text{TMP}} \) - Temporary value of DS2432 used to verify the embedded system response. The value of data inside the embedded system DS2432 if the authentication is successful.

* 384 bit user data

- \( G \) - NIST P-256 prime field elliptic curve base point for ECDH.

  * 512 elliptic curve point
    - 256 bit x coordinate
    - 256 bit y coordinate

- \( n \) - Random 256 bit integer generated by the server to multiply against the base point and generate its public key from. This value needs to be kept secret.

  * 256 bit integer

- \( z \) - Random 256 bit integer generated by the embedded system to multiply against the base point and generate its public key from. This value needs to be kept secret.

  * 256 bit integer

- \( P_D \) - Server public key generated by multiplying \( G \) by \( n \).

  * 512 elliptic curve point
    - 256 bit x coordinate
    - 256 bit y coordinate

- \( P_S \) - Embedded system public key generated by multiplying \( G \) by \( z \).

  * 512 elliptic curve point
    - 256 bit x coordinate
    - 256 bit y coordinate

- \( K \) - Resulting private key derived from multiplying other party’s public key by its own random integer. This value is used as a key for the AES channel once reduced by the Davies-Mayer Hash.
\* 512 elliptic curve point
  \- 256 bit x coordinate
  \- 256 bit y coordinate

\* MAC\_W - Message authentication code generated by DS2432 to sign the Davies-Mayer hash of the shared key K, the data stored in the DS2432, and the authentication key \( K_A \). This value signs the shared key, ensuring it came from an authentic server.
  \* 160 bit

\* C - Random integer between 0-1024 used in the public key derivation function one to determine how many iterations of SHA-1 are performed.
  \* 10 bit integer

\* MAC\_I - Message authentication code generated by embedded system DS2432 as a response and is different than MAC\_W. This is a signature of the shared key K, the authentication key \( K_A \), and the new data in DS2432 \( D_{A1} \). This value signs the shared key ensuring it came from a authentic embedded system.
  \* 160 bit

\* MAC\_R - The response from the embedded system to the server generated by PBKDF1 based on the private key K, the authentication code from the embedded system DS2432, for an iteration of C times.
  \* 160 bit

\* MAC\_RS - Expected response from embedded system given a trial value for C. Server will attempt for values of C from 0 - 1024 until a match is found for MAC\_RA or possible values are exhausted.
  \* 160 bit

\* RSP - Encrypted MAC\_RA to increase difficulty of attacker of observing the response. Information is not sensitive but there is no reason to release extra information. Helps establish that channel was created successfully.
  \* 160 bit encrypted MAC\_R.

4.4.3 Function Definitions

The following is a list of function definitions in the DA-ECDH protocol.

\* \( H_G(K_A, D_{A0}, DMH(K)) \) - Hardware Authentication function to generate a MAC to sign the DMH of the shared secret generated by ECDH. This effectively signs K with the authentication key \( K_A \) and can be verified by the embedded system DS2432.
• **DMH(K)** - An AES based hash that reduces the 512 bit ECDH shared secret into a 128 bit AES key.

• **$H_G(DMH(K), MAC_W)$** - Hardware Authentication function to generate a MAC to sign the DMH of the shared secret generated by ECDH. This effectively signs $K$ with the authentication key $K_A$ and can be verified by the embedded system DS2432.

• **$PBKDF1(DMH(K), C, MAC_I)$** - Public Key Derivation Function One which iterates SHA-1 over $MAC_I$ and $DMH(K)$ $C$ times.

• **$AES(MAC_R, DMH(K))$** - AES encryption of the result of PBKDF1, $MAC_R$ using the DMH of the shared secret $K$.

### 4.4.4 Protocol Definition

The following is a detailed run through of the DA-ECDH protocol:

(1) Initial State:

   (a) Embedded system initialized with authentication key ($K_A$), DS2432 state data ($D_{A0}$), and elliptic curve P-256 base point ($G$).

   (b) Server initialized with embedded systems authentication key ($K_A$), DS2432 state data ($D_{A0}$), and elliptic curve P-256 base point ($G$).

(2) Embedded system requests SWIP key from server initiating DA-ECDH protocol.

   (a) A random integer ($z$) is chosen and multiplied against the P-256 base point ($G$) to derive the embedded system public key ($P_S$).

   (b) Public key ($P_S$) is transmitted from the embedded system to the server.

(3) Server generates a challenge to the embedded systems request for the SWIP key.

   (a) A random integer ($n$) is chosen and multiplied against the P-256 base point ($G$) to derive the server public key ($P_D$).

   (b) The embedded system public key ($P_S$) is multiplied by the server random integer ($n$) to derive the shared secret ($K$).

   (c) A MAC ($MAC_W$) is generated to sign the DMH of the shared secret ($K$) using the DS2432 generation functionality based on the embedded system authentication key ($K_A$) and internal DS2432 data ($D_{A0}$).

      (i) The server assumes that this is the data contained by the embedded system it is communicating with.

   (d) The server transmits its public key ($P_D$) and the challenge MAC ($MAC_W$) used to sign the private key to the embedded system.
(4) Embedded system generates the shared secret \((K)\) and verifies its signature MAC \((MAC_W)\).

(a) The server public key \((P_D)\) is multiplied by the server random integer \((z)\) to derive the shared secret \((K)\).

(b) The embedded system verifies \((H_W)\) that the signature MAC \((MAC_W)\) is a valid signature of the Davies-Mayer Hash (DMH) of the shared secret \((K)\).

(i) If verified the contents of DS2432 are updated to a new state \((D_{A1})\) which is a combination of the old state \((D_{A0})\) and the Davies-Mayer Hash (DMH) of the shared secret \((K)\).

(ii) If not verified the channel is closed and no state change occurs.

(c) The embedded system has validated the authenticity of the server and the shared secret.

(5) Embedded system generates a response \((RSP)\) to the server challenge \((P_D,MAC_W)\)

(a) Embedded system randomly generates a integer \((C)\) for use in PBKDF1.

(b) The embedded system generates \((H_R)\) an internal response MAC \((MAC_I)\) to the server challenge \((P_D,MAC_W)\).

(c) The embedded system generates (PBKDF1) a response MAC \((MAC_R)\) performs a PBKDF1 on the internal response MAC \((MAC_I)\) using the DMH of the shared secret \((K)\) as a key and \(C\) as the iteration count of SHA-1 hashes to perform.

(d) The embedded system generates a response \((RSP)\) by encrypting the response MAC \((MAC_R)\) using the DMH of the shared secret \((K)\) as a key.

(e) The embedded system transmits the response \((RSP)\) to the server.

(6) The server verifies the response \((RSP)\) from the embedded system.

(a) The server updates the state \((D_{TMP})\) of its software model of the embedded system DS2432 as if the authentication was successful.

(b) The server decrypts the response \((RSP)\) using AES and the DMH of the shared secret \((K)\).

(c) The server generates the expected internal response MAC \((MAC_I)\) for the embedded system for the updated state \((DTMP)\) of the embedded system DS2432.

(d) The server generates (PBKDF1) the possible response MACs \((MAC_R)\) by performing a PBKDF1 on the internal response MAC \((MAC_I)\) using the DMH of the shared secret \((K)\) as a key for each possible iteration count \((C)\) from 0 to 1024.

(i) If a match is found

   (i.a) Embedded system is validated and secure channel that is authentic and confidential is formed.
(i.b) Server software DS2432 model data ($D_{A0}$) updated with new internal DS2432 state of embedded system ($D_{A1}$).

(i.c) Old data of server DS2432 model ($D_{A0}$) is saved ($D_{A0}^{\text{PREVIOUS}}$).

(ii) If match is not found

(ii.a) Set server software DS2432 model data ($DT_{MP}$) is restored to previous data ($D_{A0}^{\text{PREVIOUS}}$).

(ii.b) The server generates (PBKDF1) the possible response MACs ($MAC_R$) by performing a PBKDF1 on the internal response MAC ($MAC_I$), generated using previous DS2432 model data ($D_{A0}^{\text{PREVIOUS}}$), using the DMH of the shared secret ($K$) as a key for each possible iteration count ($C$) from 0 to 1024.

(e) If match found

(i) Synchronization error set DS2432 model data ($D_{A0}$) to previous DS2432 model data ($D_{A0}^{\text{PREVIOUS}}$).

(ii) No secure channel formed, retry authentication.

(f) If match not found

(i) Possibly counterfeit embedded system, take action!

(7) Secure channel that is authentic and confidential formed.

(a) Transmit SWIP key or other data.

4.5 Security Evaluation

By incorporating a minimal hardware addition in the form of the DS2432 we have been able to provide a robust yet simple authentication solution to our legacy embedded system. Also by integrating authentication with confidentiality we can establish a secure channel for transmission of the SWIP key from the key server to the embedded system. Together these solutions illuminate address the major issues of embedded system impersonation and add a layer of security to our SWIP encryption detailed in Chapter 3. Our solution however does not address the concerns about physical access to the systems. The addition of the DS2432 is just as vulnerable as if this device is removed it could be used to impersonate an embedded system. Even in a scenario where integration of the DS2432 is trivial as with application of physical protection such as potting it is unlikely that a developer could retrieve all deployed systems and apply them retroactively. This is merely a limitation of working on a legacy embedded system. In the following section we switch our focus to the evaluation of a FPGA based commodity embedded system in an attempt to illuminate many of these remaining weaknesses.

4.6 Summary

In this chapter we have:
• Identified vulnerabilities in using ECDH without authentication such as the man in the middle attack.

• Introduced the DS2432 hardware components as a secure storage solution.

• Applied the DS2432’s unique capabilities to provide mutual authentication between embedded system and key server.

• Presented a modified DS2432 Authenticated Elliptic Curve Diffie-Hellman key exchange that unifies authentication and confidentiality to prevent man in the middle attacks.

• Discussed the limitations of what security can be provided to legacy embedded systems retroactively.
Chapter 5

Securing SWIP on a Commodity Embedded Systems

The first portion of our work has focused on the securing SWIP on a legacy embedded system with a rigidly defined hardware model where our solution was limited to software-only initially. Even with the addition of hardware supported authentication there remain several vulnerabilities stemming from an attacker’s physical access to the systems at runtime. In this Chapter we move to a more flexible system model, the Field Programmable Gate Array (FPGA) based commodity embedded system. Given the flexibility of System on Chip (SOC) FPGA based designs we introduce the concept of SWIP binding to ensure that SWIP can only be used on authorized systems. Not only does this approach offer many security advantages but the application of Physical Unclonable Functions (PUFs) provides a mechanism for offline authentication, eliminating the need for a key server.

5.1 Software Intellectual Property Binding

In FPGA design, hardware intellectual property (HWIP) is defined as the soft-core (synthesized from HDL) hardware modules stored in the FPGA configuration bitstream (herein referred to as bitstream). The FPGA design can be protected by means of bitstream encryption, an option offered by several FPGA manufacturers. However, bitstream encryption is not a comprehensive solution. When the FPGA configuration contains programmable components (such as a soft-core processor), the SWIP implemented on top of that soft-core processor requires separate protection. One solution to address this issue is to encrypt the SWIP and restrict its execution to a specific FPGA. We use the term ‘Software Intellectual Property Binding’ to express this. Two components are considered in such a solution: the SWIP and the hardware platform. SWIP binding ensures that the SWIP will function only when it’s deployed on an authentic platform, which includes an authentic (designated) FPGA and a valid (designated) HWIP. Figure 5.1 illustrates that the SWIP only functions correctly when an authentic HWIP and a valid FPGA are present, such as with platform 2. Platform 3 fails because the FPGA device is not authentic, while platform 1 fails because the HWIP is not valid. The identity of a design is thus formed by the combination of a FPGA and a HWIP. In this work, we propose an end-to-end design flow for binding a SWIP to a design based on a commodity FPGA.
SWIP binding can be achieved using costly mechanisms such as secure ROM or flash memory to store FPGA specific cryptographic keys. However, this is not only expensive but rules out many commodity and legacy systems as well as being vulnerable to attack [4, 62]. In this work, we instead utilize the ability of a PUF to generate a FPGA-unique secret volatile key to achieve SWIP binding. As a PUF can be deployed securely in the fabric of a FPGA, our solution avoids the need for costly specialized hardware, and as a result, is suitable for commodity FPGA.

5.2 FPGA Based Commodity Embedded System

Similar to the legacy embedded system presented in Chapter 2 the term commodity embedded system doesn’t imply a system with a particular purpose or hardware arrangement. In the context of this work, we refer to commodity embedded system as one which posses the following properties:

- The systems hardware design relies on commodity components.
- The system is built around a commodity SRAM based FPGA.
- The system does not possess any specialized secure or cryptographic hardware components.
- The system may have basic tamper detection mechanisms.
- The system may have basic physical protection mechanisms.

Many alternative FPGA technologies exist that could be considered for the creation of a FPGA based commodity embedded system such as anti-fuse or Flash. However, each would produce a slightly different system model, as such for the purposes of this work specify that the commodity system utilizes a SRAM based FPGA. As these properties highlight
there exists a significant overlap between our legacy embedded system and our commodity embedded system. Rather than redefining our entire system model we focus on pointing out the differences between the two.

### 5.2.1 Legacy Embedded System Architecture

One of the key characteristics of a commodity embedded system is that it does not contain any specialized secure or cryptographic hardware components. Based on this property the system architecture of a commodity system directly resembles that of a legacy system as presented in Figure 2.2 in Chapter 2, Section 2.2. However in this portion of our work we specify that the system is based on a FPGA. Figure 5.2 depicts the architecture of our FPGA based commodity embedded system which is similar to that of our legacy embedded system. The primary distinction is the presence of the FPGA and the shift from standalone processor to a soft-core processor contained within the FPGA’s fabric. This provides a great deal of flexibility and customization as the FPGA fabric and soft-core processor can readily be modified as they are programmed entities synthesized from HDL. In our solution we deploy a PUF based mechanism in the FPGA fabric as a secure key storage mechanism.

![Figure 5.2: The architecture associated with a FPGA based commodity embedded system.](image)

### 5.2.2 System Components

The FPGA based commodity embedded system architecture introduced in Figure 2.2 contains nearly identical system components to the legacy embedded system. The only differences are noted as follows:

- **Hardware** - The physical silicon and system (PCB, enclosure, etc) on which the SWIP is deployed for storage and processing. A systems hardware is comprised of several other components:
* FPGA - The physical silicon that the HWIP and SWIP are deployed on. It might contain a hard core processor, other specialized hardware such as multipliers, and internal RAM blocks.

* HWIP - The soft core processor and other hardware components, including the PUF, which are configured into the FPGA fabric. This is typically the level at which HWIP protection(binding schemes are implemented, for example with bitstream encryption [70]. The combination of FPGA and HWIP replaced the embedded processor from the legacy embedded system presented in Chapter 2.

- Software - The same combination of Software Intellectual Property (SWIP), Non Software Intellectual Property (NON-SWIP), and third party software, exist on a FPGA based commodity embedded system.

5.2.3 System Interests and Trust Boundaries

The parties with interests in our system remain similar to that of our legacy embedded system. Specifically there are two changes of note: First, the hardware manufacturer is responsible for manufacturing of the FPGA. Second, the system developer is also responsible for developing the HWIP that operates on the FPGA. The remaining parties of the SWIP developer, end user, and adversary remain the same. The trust relationship and model for these parties also remains similar as denoted by Figure 5.3, as does our adversary and their capabilities.

5.2.4 Development Design Flow

The design flow of a FPGA based commodity embedded system remains very similar to that of a legacy embedded system. Both begin with hardware development and transition into system development where a platform for software development is established. However, due to the configurable nature of FPGA based designs a secondary system development step typically occurs in tandem with software development. In this stage the FPGA design is developed, often in a closely integrated manner with the software, before both are deployed on the hardware system. Software development remains unchanged as does its deployment. The key difference occurs during startup where an FPGA configuration bitstream first programs the FPGA with the soft-core processor and other components. After this the execution process continues as if on a typical embedded processor. As the target of our work is a commodity system, which we define as having no specialized hardware components, we can employ a more rigid design flow. This allows us to focus our efforts on the software and FPGA development, deployment, and system use as shown by the shaded regions in Figure 5.4.
Figure 5.3: Trust boundary and relationships between system components and development parties. Dashed regions which overlap signify trusted relationship between parties.

Figure 5.4: System design flow from hardware development through end use, shaded regions highlight primary focus of work.
5.2.5 Security Evaluation

A FPGA based embedded legacy system offers many useful features from a security standpoint such as system on chip designs, configurability, and bitstream obfuscation. While we take advantage of these features the attack scenarios remain the same for the initial system evaluation. However, application of these features helps mitigate some security vulnerabilities discussed in Chapter 3, Section 2.6.2. First, the nature of FPGAs allow for the implementation of several components within the FPGA fabric such as a ROM. More importantly such components only have internal connections between each other, limiting the ability of an attacker to observer or inject information on their busses. Second, the configurable nature of FPGAs, allow us to disable selective hardware features for internal components such as JTAG based debugging. Finally, the configuration of the FPGA fabric is complex and not an open standard. This adds an inherent layer of obfuscation to designs deployed on the FPGA. Again we identify the most critical vulnerability as the storage of SWIP in an unprotected format in non-volatile memory.

5.2.6 Related Work

This work relies on a PUF as the root of trust. A PUF is a platform-unique function which, when supplied with an input challenge, produces an output response. The response is determined by the behavior of a complex, unclonable physical system, such as the delay variation of logic and interconnects in an FPGA due to manufacturing process variations. It can be used to authenticate chips and generate a volatile secret key required for cryptographic operation without the need of an expensive non-volatile memory [65]. It is also useful in SWIP protection [28] as well as in securing private information in many applications.

Several different types of PUF have been proposed so far. A ring-oscillator (RO) based PUF [65] is of particular note among them because of its easy implementation on the FPGA. The complex nature of an FPGA [20] provides a platform to deploy traditional forms of intellectual property protection such as tamper-resistance and obfuscation. By coupling a PUF with these techniques, we can provide an efficient platform for binding SWIP to an FPGA. In the work of Guajardo et al. [29], a SRAM-based PUF protection mechanism is proposed for securing HWIP modules. In contrast, our work focuses on protecting SWIP and provides a demonstration system that explores generation and encryption of the protected SWIP. In addition, we also provide a detailed mechanism that addresses how to perform the parsing, decryption, and loading of encrypted SWIP sections. Guajardo assumes the existence of such a hardware mechanism and does not go into detail about the nature of such a component. An updated Aegis architecture has been proposed in [66] for secure software execution using PUF. That work proposes the use of PUF for runtime memory-integrity through the use of hash trees. Our approach addresses configuration of SWIPs, and shows how to authenticate them onto an FPGA fabric.
5.3 Overview

Figure 5.5 illustrates how we employ our PUF based SWIP binding methodology in the context of FPGA based system design. The elements in gray represent the major contributions of our approach and are broadly divided in two parts:

- Before delivery to the end-user, inside the trusted and secure development environment, an FPGA-unique key is extracted from the PUF in a process called enrollment. The SWIP is then bound to the FPGA by encrypting it using the PUF key with the help of a custom encryption tool.

- After delivery to the end-user, when the FPGA boots up, a security kernel (SK) extracts the PUF-based key from the FPGA to decrypt the encrypted SWIP for execution.

The FPGA, its configuration bitstream (including the PUF and HWIP) and the protected binary (including the encrypted SWIP and the SK) constitute the final product to the end-user. It is critical to prevent an attempt to modify the components of the delivered product for the purpose of extracting the PUF key. To solve this problem, we implemented an integrity mechanism which is discussed in detail in Section 3.E.

5.4 PUF Based Key Storage

A Physical Unclonable Function (PUF) is a circuit that is used to extract or measure a physical characteristic on the hardware component on which it is deployed. For instance given
a set of FPGA’s we could implement a simple circuit such as a ring oscillator. A ring oscillator being an odd series of invertors whose output is connected to its input. Once triggered the circuit oscillates freely until the feedback is broken. Due to process variations, beyond the control of the hardware manufacturer, the oscillators on each FPGA would oscillate at a different frequency. By exploiting these random but static manufacturing process variations in ring oscillator frequencies we can create a challenge response mechanism that is capable of identifying a particular device. A great advantage of this approach is that the challenge and response mechanism functions as a property of the physical characteristics of a particular device. As such the same circuit is employed on every device and yet produces a unique challenge/response pair for a particular device. A common mechanism for extracting an identifier from a ring oscillator based PUF’s is to perform a pair-wise comparison. Given a series of ring oscillators on a chip, a challenge is used to select a pair to be enabled for a fixed period of time. A counter keeps track of the frequency of which each oscillates and a single bit is generated based on which is faster. Combining several comparisons can be used to generate an arbitrarily long response, which can be utilized as a key.

5.4.1 Error Correction

PUF outputs are noisy by nature; a finite number (though low) of PUF output bits vary over time. However, for cryptographic operation, a stable key is necessary. Generating noise free keys from PUF is an active area of research, and several error correction schemes have been proposed such as [43]. Most of them require a complex implementation in software and/or hardware. The main objective in this work is to show the use of a PUF in the SWIP protection mechanism, and therefore, we have implemented a simple but effective error correction mechanism. By only selecting the RO pairs that have a relatively high difference in frequency we can extract a stable key and ignore error prone RO pairs. The drawback of this scheme is that we are not able to fully utilize all the ring oscillators for key extraction. However, other error correction techniques can be employed in a production scenario.

5.4.2 Security

A major concern with the new application of the PUF for remote authentication is that each exchange exposes a CRP of the PUF. As the PUF only contains a finite (sometimes quite limited) number of CRP’s an attacker can attempt a model building attack. Several proposals have been introduced for preventing such attacks but they each rely on disrupting the relationship between input and output of the PUF. Gassend [24] proposes the application of a random secure hash function on a pre-challenge before it is passed to the PUF and on the response after it is generated. Due to the principles of a secure hash, hashing the pre-challenge before it is issued to the PUF makes it difficult for an attacker to specify the challenge the PUF receives. Similarly passing the response through a hash hides the value of the response from the attacker due to the one way nature of hashes. In this way such a
scheme removes a direct observable relationship between the challenge and response outputs of the PUF. Such functionality can greatly improve a PUF’s resilience to attack and is trivial to implement as software interface layer in our SK.

5.5 Software Intellectual Property Binding

The binding mechanism that ties SWIP to a single physical FPGA and HWIP is the use of a PUF derived key to encrypt the SWIP. In such a way only a system that can generate the correct key can decrypt the SWIP and execute it. Despite the distinct mechanisms used for key storage the actual encryption of the SWIP in our solution is nearly identical to the one employed on our legacy embedded system.

5.5.1 Encryption

As in our legacy solution, our commodity solution requires the presence of a Security Kernel (SK) to retrieve the PUF based key and decrypt the SWIP. The result is a system that contains two major types of software. One consists of SWIP and its supporting code which must be encrypted and the other the SK which remains in plain text. The C section attribute in conjunction with a naming scheme is used to denote code sections belonging to the SK. Figure 5.6 shows an unprotected binary containing SWIP and our SK in the ELF format. As with our previous solution we take advantage of objdump to obtain a detail list of the binaries contents. This includes the sections contained in the binary and their attributes such as name, file offset, and size. Based on the header information and our naming convention, our Intellectual Property Encryptor (IPE) utility retrieves the SWIP section from the binary file. Each section is then encrypted by the IPE using 128-bit AES in counter mode and a key derived from the PUF enrollment process.

Besides the encryption of software sections, the IPE also creates an additional data section in the resulting encrypted binary. Space for this data section is allotted in the security kernel and holds the offset, size, and decryption information for each software section that was encrypted. This data section is used by the Security Kernel at boot to locate encrypted software for decryption and deployment to its proper location. Unlike our legacy solution the nature of the cryptographic information stored by the IPE is drastically different. First, we remove the need for a nonce by procedurally generating the value based on the section hash and position. Second, preceding encryption we perform a hash of the plain-text section to generate a hash digest. This digest allows us to verify the decryption and validity of the section after decryption not only to protect against incorrect keys but malicious code sections. After the IPE concludes the resulting binary will contain both encrypted and plain text code sections. Any standard programming utility may be used to deploy the protected binary to its target system. Encryption of the Elf binary by the IPE and its deployment all occur within the trusted development environment and our not subject to attacks.
Figure 5.6: Encryption of an Elf binary by the IPE.
5.5.2 Decryption

At boot time, when the protected binary is downloaded to an FPGA, the SK performs the operations of the IPE in reverse as shown in Figure 5.7. Utilizing the challenge stored in it by the IPE, the SK retrieves the PUF key. Next the security kernel parses the encrypted binary using the header information included by the IPE. As each section is decrypted using 128-bit AES and the PUF derived key, the SK copies the decrypted contents to their targeted internal memory location. During this process the SK hashes each section and compares the resulting digest to the one stored in the SK by the IPE. If the hash values differ this could indicate a compromised section or a failed decryption. After decryption is completed and validated, the SK turns execution over to the SWIP. If all sections of SWIP are decrypted then the memory occupied by the SK can be reallocated for other uses.

![Figure 5.7: Decryption of software sections by the SK at boot.](image)

5.6 Trusted Boot

A key concern during the boot procedure is maintaining the integrity of the SK and ensuring that only its validated software executes. This is necessary to ensure that the system boots in a trusted state and requires that any tampering to the boot or interrupt vectors as well as to the SK software be detected. However, the drawback of our software based scheme is that the SK can’t be trusted because a plain text binary can be read and modified. As the security kernel is not considered as SWIP, its confidentiality is not
of concern. The greater issue is that a compromised security kernel could be utilized to retrieve the PUF key, and the decrypted SWIP. We address this issue with the inclusion of an Integrity Kernel (IK).

5.6.1 Integrity Kernel

Validation of the security kernel is the primary function of the IK. A hashing algorithm is commonly used to establish the validity of software by comparing the results of the hash with a reference value. We boot our system in the IK which runs a hash on the security kernel and validates it against a reference result. By verifying that it has not been tampered with, the execution can pass to the security kernel, and then it can begin decryption of the SWIP. It is imperative that the IK and the boot procedure are secure against attacks. If an attacker can bypass either the IK or SK they could potentially execute untrusted software and extract the key from the PUF. To achieve a trusted boot (Figure 5.8), we introduce an obfuscated ROM to provide a tamper resistant storage mechanism for the IK and all values needed to control the boot process. This prevents an attacker from subverting or surpassing the IK which in turn verifies that the SK has not been modified.

![Figure 5.8: Trusted boot procedure through IK verification of the SK.](image)

5.6.2 Obfuscated ROM

An FPGA bitstream is believed to have an inherent layer of obfuscation. Though LUT configuration and BRAM contents can be accessed relatively easily, a complete reverse engineering of the bitstream into a net-list has not been reported so far [20]. Based on this assumption, an obfuscated ROM is implemented using multi-level logic in the FPGA avoiding direct storage in LUTs or BRAMs. This ROM is used to securely store the integrity-kernel (IK) binary. Even though the required logic circuits are implemented using LUTs in an FPGA, the contents of the ROM cannot be extracted just by reading the contents of the LUTs. This is because the ROM circuit is formed by a combination of several LUTs which
is spread over the FPGA, and interleaved with other circuit components. The IK binary
includes the IK software, the boot vectors, and the interrupt vectors. The vectors are in-
cluded into this binary to help protect the software execution flow. These values are hard
coded as the content of the obfuscated ROM. Since synthesizing the obfuscated ROM is
trivial, modifying the design of the integrity kernel is not difficult. This helps to maintain
the flexibility of our design flow. However, it is important to maintain a low footprint while
implementing the ROM.

5.6.3 Hashing Algorithm

Employing an obfuscated ROM is costly in terms of area and is primarily why it is not
used to deploy the SK. As a result, selection of a compact hashing algorithm with small
memory footprint is essential. Traditional hashing algorithms such as SHA-1 are large due
to the size of their internal state. An alternative solution is the use of a cipher based Davies-
Mayer hash which allows us to leverage the compact nature of certain ciphers. XXTea for
example can be deployed in under 380 bytes in such a configuration [48, 68]. Combined with
initialization data and the expected results f the hash, we are able to implement it in a 512
byte block of obfuscated ROM.

5.7 Design Summary

All the software and hardware components, required for our design flow, are illustrated
in Figure 5.9 below from a developer’s perspective with shaded figures as main components
of our design flow. Our proposed system is able to achieve a high flexibility for several
reasons. First, we do not rely on any specific hard core facilities or capabilities in an FPGA.
Our design only requires the ability to deploy a soft core processor. Only the interface with
the hardware PUF would be system specific. Second, by maintaining the majority of our
functionality in software, we ensure rapid substitution of components such as the PUF, error
correction, and various cryptographic primitives to meet the specific needs of the developer.
Finally, by developing our design using standard C libraries we ensure compatibility across
a wide array of soft and hard core processors.

5.8 Security Evaluation

At first, we assume that the developer’s environment will protect both HWIP and SWIP
sources, including the PUF design (as defined in Figure 5.3). Based on this assumption, we
make an effort to achieve SWIP binding in the untrustworthy user environment. The goal
of the attacker is to reveal the secret key in order to decrypt the SWIP. In our method, the
PUF-based key remains internal to the FPGA and never gets exposed, so the attacker has
to try to modify either the software or the hardware platform consisting of the FPGA device
and the configured HWIP in order to determine key. Based on this assumption, we discuss
Figure 5.9: Design flow overview. Shaded blocks represent additions or modifications.
a few relevant hardware and software security issues.

5.8.1 Hardware Analysis

If a physical attack is mounted on the FPGA device such as by laser cutting or removing chip layers, it is believed in current literature that the complex and sensitive delay behavior of the PUF changes and the key is destroyed. As our system does not utilize a specialized bitstream protection scheme, the bitstream containing the PUF is visible to an attacker. However, due to the complex nature of an FPGA bitstream; it possesses an inherent layer of obfuscation. Moreover, we avoided use of easily accessible components like LUTs and BRAMs to store sensitive information making it difficult to extract useful information from the bitstream or to modify it [20]. This is comparable to the obfuscation employed on software binary to prevent reconstruction of source code (in this case HDL) [25, 47]. As long as these assumptions can be made concerning the nature of the FPGA bitstream, adequate security is available to prevent an attacker from separating our protection mechanisms from our PUF implementation. Additionally, when combined with a fuse based FPGA the bitstream can’t easily be extracted and the device can’t be reprogrammed. Even if the attacker could extract the PUF design it would be impossible for them to redeploy a compromised design to the same physical FPGA. As the PUF design does not store the key but extracts it from the physical operation of the FPGA fabric this would prevent an attacker from being able to retrieve the key.

5.8.2 Software Analysis

As the SWIP is hashed after decryption to verify its integrity it is very difficult to modify the protected binary in any useful way. On the other hand, since the SK is in plain text, it could be modified. However, the addition of the IK using the obfuscated ROM prevents the execution of the SK if it has been altered. An attacker could attempt to subvert a deployed system in such a way that they could execute malicious code to extract the key. We prevent such an attack by the application of obfuscated ROM inside the bitstream which contains the boot/interrupt vectors for the processor as well as the IK. The IK stored in the obfuscated ROM verifies the integrity of the SK before it is allowed to execute. Likewise the SK decrypts and verifies the integrity of SWIP before it allows it to begin execution. The only way an attacker can then subvert this protection is by successfully reverse engineering and altering the bitstream in a meaningful way, attacking the implementation of the SWIP (buffer overflow, etc.), or directly attacking physical weaknesses of system hardware. However, we note in the discussion of our trust model and security analysis that we assume that the bitstream is obfuscated, the SWIP is well written (not prone to vulnerabilities), and that the FPGA is tamper resistant. Finally, there is the concern for traditional methods of attacking software at runtime such as buffer overflow or exploitation of inherent weaknesses in the software. In general, SWIP binding can do little to avoid such issues. Rather, the SWIP that is being protected must be validated as being well written to avoid such problems.
After the initial secure boot, no guarantees can be made to the state or integrity of the SWIP at runtime.

5.8.3 System Design Tradeoff

As our system is designed to provide a flexible framework for SWIP binding we only specify the general functionality of certain components such as the PUF error correction. It is important that the implementations selected for these system components provide adequate security. Specifically error correction schemes should not leak information about the key and maintain a good source of entropy. Similarly the output of a PUF should be unique across devices and show a high level of stability [65].

5.9 Summary

In this chapter we have:

- Provided an overview of our SWIP solution for FPGA based commodity embedded systems.
- Defined a system model for a FPGA based commodity embedded system.
- Introduced a PUF as a securing storage mechanism for SWIP keys on an FPGA based system.
- Highlighted the differences between SWIP encryption and binding.
- Demonstrated the concept of a secure boot.
- Introduced our Integrity Kernel that handles verification of the SK.
- Performed a security evaluation of our SWIP security solution for FPGA based commodity embedded systems.
Chapter 6

Portable Embedded Software Intellectual Property Security

Our exploration of securing SWIP on legacy and commodity systems has yielded two solutions explored in Chapters 3 and 5. While these solutions are independent there exists a significant overlap between their components and functionality. In this chapter we introduce the Portable Embedded Software Intellectual Property Security (PESIPS) system to provide a flexible basis from which such solutions could be developed.

6.1 Overlap in Securing SWIP on Embedded Systems

Figure 6.1 provides a side by side comparison of the modified design flow utilized in our SWIP security solutions for legacy and commodity systems. As illustrated there are significant similarities in the software modifications for both systems. Specifically these are the attribute based identification, security kernel, and intellectual property encryptor utility. The primary deviation in the designs comes from the storage mechanism for the SWIP key, which while critical to the systems functionality, is easily isolated. As such we can focus on the shaded components in Figure 6.1 in providing a general and easily portable basis for securing SWIP on an embedded system. The absence of an integrity kernel in the legacy system can be attributed to the lack of any secure storage mechanism integrated into the embedded processor. However, to accommodate the widest possible range of applications we include it along with the three other noted components.

6.2 PESIPS System Overview

Figure 6.2 illustrates the generic design flow for the deployment of PESIPS. The elements in gray are those integral to the PESIPS system while those in white are solely the system or software developer’s responsibility.

- IP Encryptor (IPE) - Utility used to perform the encryption and configuration of the binary containing the PESIPS module. The IPE is responsible for embedding all information needed by the PESIPS module to perform boot-time decryption of the SWIP.
Figure 6.1: SWIP security design flow overview for legacy and commodity systems.
• PESIPS Module - The collection of components including integrity kernel, security kernel, supporting cryptographic algorithms, and storage interface that are responsible for the boot time decryption and verification of SWIP.

![Diagram of PESIPS system deployment](image)

Figure 6.2: Generic design flow for deployment of the PESIPS system. Shaded figures denote PESIPS system components.

Typical applications of the PESIPS system should require minimal design considerations and modifications from the system and software developers. These include:

- Selection of a cryptographic suite including a block cipher and cryptographic hash.
- Application of naming conventions to software that supports the PESIPS module.
- Initialization of trusted boot procedure.
- Integration of PESIPS storage interface and system storage solution which securely contains SWIP keys.
- Proper configuration of PESIPS module.

The requirements and guidelines for the design considerations are further discussed in Section 6.5.

### 6.3 Intellectual Property Encryptor

Unlike our solutions for the legacy and commodity embedded systems we desired a more flexible approach to encrypting the binary and configuring the PESIPS module. As such we introduce a two phased approach to processing an ELF binary as shown in Figure 6.3. First, the IPE reads the binary, extracting the section and other information from the file to generate a set of configuration files. The software or system developer can then customize these files to more granularly adjust the PESIPS module. Finally, the IPE is re-run with the customized configuration files to drive the encryption of the ELF binary and the configuration of PESIPS module. This is accomplished through an easily customized shared cryptographic library further discussed in Section 6.5.
Figure 6.3: Iterative approach to the encryption and configuration of ELF binary by PESIPS system IPE.
6.3.1 Generating Configuration Files

One of the major key improvements of the IPE over either previous solution is the removal of a third party utility for parsing the ELF binary structure. Instead we have incorporated native ELF support through the use of the GNU Binary File Descriptor (BFD) library which is included in the GNU compiler collection. This library allows us to directly open and parse an ELF and extract the critical section information. As such we no longer have to account for variations in the output of different versions of objdump or similar utilities. Another key advantage of employing the BFD library is that over 25 processor architectures and 50 file formats are currently supported by it. While our solution is specifically geared to ELF format binaries at this time it does allow for easy expansion to other formats. After the IPE parses an ELF binary it generates a pair of configuration files: a general configuration file and a cryptographic support file. The general configuration file (.cfg) contains the information extracted from the binary through the BFD as well as default settings including:

- File Name - Name of the file that the configuration file belongs to. This must match for application of the configuration to the binary. Mismatched files could result in a binary that is not usable.

- File Details
  - Start Address - Entry point of the binary, typically the PESIPS system should begin execution with the integrity kernel.
  - Size - Size of the binary file.
  - Sections - Number of data and code sections contained in the binary.
  - Hash - Hash digest of entire file contents used to verify binary file for application of the configuration file.

- Section Details - A list of each section in the binary and its associated properties and configuration options.
  - Index - Index of section as located in the binary.
  - Section Name - Name of the section, checked to determine section type based on a naming convention. The following prefixes are reserved for our implementation:
    - .pesips_genreal - Sections of the PESIPS module or supporting code. These sections are stored in plain text and may or may not require authentication depending on the nature of the system.
    - .pesips_config - Storage location for the PESIPS configuration information that is used at boot by the PESIPS module to authenticate and decrypt the SWIP.
    - .pesips_intk - Special section of the PESIPS module that only contains the integrity kernel. This is stored in plain text and does not require authentication.
- * - All other software sections, including SWIP that must be encrypted.

* Size - Size of the section contents.

* Offset - Offset in the binary file of where the contents of the section begin. Used to locate section corners for encryption or modification during encryption and configuration.

* Address - Target location of the section in the processor memory map after boot and decryption.

* Key - Option that specifies the index of the key that will be used to encrypt this section, default value of zero denotes no encryption for the section.

* Type - Section type to verify intended section role, sections are typed based on naming convention. Secondary option that allows developer to add sections to .pesips_general manually without having to individually assign naming convention.
  - p - PESIPS general component.
  - i - PESIPS integrity kernel.
  - * - Other software section.

* Hash - Option to specify if a hash of the section will be generated.
  - y - Generate a hash for the section.
  - n - Do no generate a hash for the section. with the key support and acts as an interface to system storage.

The cryptographic support file (.crypt) contains the key information needed to encrypt the binary during configuration including:

- File Name - Name of the file that the configuration file belongs to. This must match for application of the cryptographic support to the binary. Mismatched files could result in a binary that is not usable.

- File Details
  - Start Address - Entry point of the binary, typically the PESISPS system should begin execution with the integrity kernel.
  - Size - Size of the binary file.
  - Sections - Number of data and code sections contained in the binary.
  - Hash - Hash digest of entire file contents used to verify binary file for application of the cryptographic file.

- Key Information - Key information needed by the cryptographic library block cipher to be initialized.
After the configuration files are created and customized, encryption and configuration of the ELF binary including the PESIPS module can occur. During the creation of these configuration files the IPE will check for a series of requirements to ensure that the proper operation of the PESIPS system can be ensured. We cover these configuration requirements further in Section 6.5.

6.3.2 Encryption and Configuration

Given a configuration file, cryptographic support file, and ELF binary the IPE will first begin by verifying the system setup in two steps. First the information about the ELF binary stored inside the configuration and cryptographic support files are verified against the provided binary. This is accomplished by checking the file name, file details, and most importantly the hash of the file. Once these are confirmed the IPE will perform a secondary check against a list of requirements, discussed further in Section 6.5, displaying any errors or warnings. After verification the IPE begins the encryption process, which is nearly identical to the one presented in our commodity system solution and is depicted in Figure 6.4. Utilizing the configuration file, the IPE extracts the contents of a section from the binary. Based on the key index specified in the configuration file for the section the IPE uses the corresponding key from the cryptographic support file. If the key file has not been provided or the specific key not set, the IPE will utilize a key generator supplied by block cipher and updated the cryptographic support file. At the same time the plain text contents of the section are fed into the hash engine to generate a verification digest.

Once the encryption process has been completed on all sections marked for encryption the IPE populates the configuration storage section in the PESIPS module. This contains all the necessary information needed to decrypt the binary file at boot and includes:

* Index - Index of the key starting at index zero which is considered the root key for the configuration storage. If a negative index is specified then there is considered to be no encryption. By default a blank root key is included in the cryptographic support file.

* Key Material - A blob of initialization parameters and key material needed by the block cipher. No key material present indicates that this should be generated during encryption and configuration by the IPE and is not supplied by the developer.

* Key Support - Information that may be needed to extract or retrieve the key. This can be such assisting information as a challenge for a PUF based key storage system.

* Load Flag - Denotes if a fetch operation must be called in order to retrieve a key. This is used in conjunction with the key support and acts as an interface to system storage.
Figure 6.4: PESIPS based encryption of a binary file.
• Root Information - Information used to decrypt the remaining configuration storage information, it is always assumed that his key must be retrieved from system storage.

  * Root Key Support - Information that may be needed to extract or retrieve the key or assist the block ciphers operation. This can be such assisting information as a cryptographic nonce or a challenge for a PUF based key storage system.
  * Configuration Digest - Cryptographic hash digest of the secured configuration storage to ensure it hasn’t been tampered with.

• Secured Configuration Storage - Information used to decrypt the encrypted binary and is encrypted it’s self by the root key.

  * Header - Minimal set of information about the encrypted section. One header is stored for each section.
    – Offset - Starting location of the section in the system memory map before decryption.
    – Size - Size of the section
    – Address - Destination of decrypted section.
    – Key Index - Index of key used by block cipher to decrypt the section.
  * Digest - The digest of the decrypted section. On digest is stored for each section.
  * Key Information - Additional key information for any subsequent keys used for SWIP decryption or other system tasks. A single Key Information is stored for each key.
    – Index - Index of the key.
    – Load Flag - Denotes if a fetch operation must be called in order to retrieve a key. If the load flag is set then the key support is used to retrieve the SWIP key from system storage. If not then it is treated like key material and used directly as a SWIP key.
    – Key Support / Material - Information that may be needed to extract or retrieve the key. This can be such assisting information as a challenge for a PUF based key storage system. At this point the IPE has finished and produced an encrypted ELF binary with a configured PESIPS module which is ready to be deployed to an embedded system.

6.4 PESIPS Module

Figure 6.5 depicts the PESIPS module and its constituent components where the arrows indicate elements provided by the system or software developer. At boot execution is transferred to either the integrity kernel or security kernel depending on the nature of the system (availability of features like obfuscated ROM). Once the integrity kernel validates the security kernel execution is transferred to it. The SK parses the contents of the configuration
storage and decrypts each SWIP section by retrieving the SWIP utilizing the system specific storage plug-in. After decryption is completed execution is transferred to the SWIP and the memory space occupied by the PESIPS module can be re-tasked.

Figure 6.5: PESIPS module deployed onto embedded system to authenticate and decrypt SWIP at boot time.

6.4.1 Integrity Kernel

Validation of the security kernel is the primary function of the IK. As with our commodity solution, we use a hashing algorithm to validate the security kernel. However due to the tightly integrated nature of the IK and its base hash as well as the often tight space requirements the hash engine from the cryptographic shared library is not used. Instead we maintain a separate and minimal hash function such as XXTea for these purposes. First the contents of the SK are copied into trusted memory then they are verified before execution is transferred. This trusted boot process through the IK is shown in Figure 6.6.

6.4.2 Security Kernel

The major modification to our security kernel and decryption process over our commodity solution is the inclusion of a root key. This functions to protect the configuration information stored by the IPE inside the PESIPS module from modification. As the security kernel is given execution it begins by using the root key information to extract the root key from system storage. Next, the SK decrypts the configuration information and verifies it against the configuration digest. This process is similar to that of decryption and verification of sections as shown in Figure 6.7. However, the root key and configuration digest are stored in a fixed location, hard coded into the SK. After the SK has decrypted and verified the configuration information, decryption can continue for the SWIP. This is performed by the
<table>
<thead>
<tr>
<th>Secure ROM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Boot Vector</strong></td>
</tr>
<tr>
<td>goto Integrity_Kernel()</td>
</tr>
<tr>
<td><strong>Interrupt Vectors</strong></td>
</tr>
<tr>
<td>Integrity_Kernel()</td>
</tr>
<tr>
<td>copy_Security_Kernel();</td>
</tr>
<tr>
<td>result = XXTea();</td>
</tr>
<tr>
<td>if( result != expected )</td>
</tr>
<tr>
<td>restart;</td>
</tr>
<tr>
<td>goto_Security_Kernel();</td>
</tr>
<tr>
<td><strong>Expected Hash Results</strong></td>
</tr>
</tbody>
</table>

Figure 6.6: Trusted boot procedure through IK verification of the SK.
SK parsing each configuration entry to locate the encrypted SWIP section and then decrypted with its corresponding key. One of the benefits of utilizing the root key approach is that subsequent keys may be stored in directly in the configuration information instead of the system storage. The “load flag” denotes this arrangement, where if set then the key must be retrieved from the system storage. If the flag is clear then the SWIP key is stored directly in the configuration information. Once decrypted, each section is then verified against it stored digest. This process repeats until execution is turned over to the SWIP after all sections have been decrypted and verified.

6.4.3 Component Management

Cryptographic algorithms have a wide range of applications beyond just SWIP security such as securing communications. As such many of the algorithms provided by the cryptographic library may be of use throughout the system life. The component management run-time API simply provides an interface to these cryptographic elements. We provide this interface to ensure that there is no accidental release of sensitive key information. The component management module enforces this by wiping any state information belonging to the block cipher or secure hash when it is called from outside of PESIPS.

6.5 PESIPS Requirements

The PESIPS system, specifically the PESIPS module is capable of providing a layer of security around a system’s SWIP. However, this is only possible if: the proper cryptographic algorithms are selected, naming conventions are adhered to, trusted boot is ensured, proper integration of system storage is achieved, and the PESIPS module is configured correctly. In this section we explore some of the necessary considerations needed to ensure that the PESIPS module can function to its fullest.

6.5.1 Shared Cryptographic Library

The PESIPS system is designed to meet a wide array of applications and as such is designed to be flexible. Based on this principle we have designed the system to support a wide array of block ciphers and cryptographic hash functions. This allows a developer to trade off security, performance, and software footprint as they optimize their application. In order to take advantage of this feature a developer need only ensure that the algorithm supports a minimal set of functions required by the PESIPS shared cryptographic library API. For the block cipher this API includes the following functions:

- `#define CIPHER_BLOCK_BYTES`
  * Size of the cipher block in bytes.
Figure 6.7: PESIPS decryption of an encrypted system through the security kernel, note the developer provided system storage.
typedef struct cipher_key;
  * Data blob that contains cipher key in appropriate format.

typedef struct cipher_state;
  * Data blob that contains the cipher state information.

int cipher_initEncrypt( const cipher_key *key, cipher_state *state );
  * key - Key information used to initialize the state information.
  * state - State information of the cipher.
  * returns - 1 on success, 0 on failure.
  * Initializes the block cipher for decryption with the given key and stores its state information in state.

int cipher_initDecrypt( const cipher_key *key, cipher_state *state);
  * key - Key information used to initialize the state information.
  * state - State information of the cipher.
  * returns - 1 on success, 0 on failure.
  * Initializes the block cipher for encryption with the given key and stores its state information in info.

void cipher_clean( cipher_state *state );
  * info - State information of the cipher.
  * Cleans the cipher state information to help avoid leaking SWIP keys.

int cipher_encrypt( uint8 *input, uint8 *output, const cipher_state *state );
  * input - Pointer to the state of one block of input.
  * output - Destination of one block of the encrypted input.
  * state - State information of the cipher.
  * returns - 1 on success, 0 on failure.
  * Encrypts a block of data at input and stores it at output. Supports output and input being the same memory location.

int cipher_decrypt( uint8 *input, uint8 *output, const cipher_state *state );
  * input - Pointer to the state of one block of encrypted input.
  * output - Destination of one block of the decrypted input.
  * state - State information of the cipher.
* returns - 1 on success, 0 on failure.
* Decrypts a block of data at input and stores it at output. Supports output and
input being the same memory location.

- int cipher_generateKey( const cipher_key *key );
  * key - Key to randomly generate.
  * returns - 1 on success, 0 on failure.
  * Randomly generates a cipher

- int cipher_test( );
  * returns - 1 on success, 0 on failure.
  * Cipher test against a known test vector, hard coded into the test function.
    Only available if “PESIPS_DEBUG” is defined in the pre-processor. This is to
    validate functionality in both the IPE and PESIPS module.

For the cryptographic hash this API includes the following functions:

- #define HASH_BLOCK_BYTES
  * Size of hash block in bytes.

- typedef struct hash_digest;
  * Data blob containing the hash digest.

- void hash_init( hash_digest *digest );
  * digest - Hash digest state.
  * Initialize a hash digest with any necessary state.

- void hash_clean( hash_digest *digest );
  * digest - Hash digest state.
  * Clear out a hash digest.

- void hash_hash( uint8* input, hash_digest *digest, uint32 lengthBytes );
  * input - Input data to be hashed.
  * digest - Digest state to store results.
  * lengthBytes - Length in bytes of the hash input.
  * Hash a variable length block that is a multiple of bytes in length this should
    include all necessary padding.
• void hash_round( uint8* input, hash_digest *digest );
  *
  * input - Input data to be hashed.
  * digest - Digest state to store results.
  * Hash one block of data.
• int hash_compare( const hash_digest *digest1, const hash_digest *digest2 );
  *
  * digest1 - Digest state.
  * Digest2 - Digest state.
  * returns - 1 if equal, 0 if not.

6.5.2 Application of Naming Conventions

During system and software development, especially during integration of the PESIPS module it is important to identify and name sections correctly. If software needed to initialize the key storage for instance is incorrectly labeled then retrieval of the SWIP may not be possible. Also incorrectly labeled SWIP could avoid being encrypted and as a result be exposed to attackers.

6.5.3 Trusted Boot

Execution of a system with the PESIPS module should begin in a small code stud that disables interrupts before turning execution over to the integrity kernel for verification of the security kernel. Interrupts can be again enabled after decryption of all SWIP has completed. In addition is necessary that the developer can guarantee that the IK and boot vectors can’t be modified through application of a secure memory such as obfuscated ROM. If there is sufficient storage for the entirety of the SK and supporting software then the IK may be removed and boot may directly transfer to the SK.

6.5.4 System Storage Integration

Similar to the flexibility offered by our cryptographic library API system storage also takes on a very simple interface to ensure compatibility with a wide array of possible implementations. This includes the PUF based or server based SWIP key storage solutions. The following interface is required by the PESIPS module configuration storage:

• typedef struct system_keySupport;
  *
  * Data blob containing the key support information.
• int system_retrieve( uint32 index, system_keySupport *ks, cipher_key *key );
When decrypting a section if the load flag is set then the security kernel will attempt to retrieve the key from the system storage using this API. The security properties of the PESIPS system are directly linked to the robustness of the system storage. An ideal solution would be an internal storage mechanism such as that provided by a PUF in our commodity system. However, a server based solution is still applicable with the interface. The software responsible for interface with the system storage must be labeled as “.pesips_general_” to avoid encryption so it’s available to the SK at boot.

6.5.5 PESIPS Module Configuration

The configuration of the PESIPS module through the IPE may yield different types of warnings or errors if options are selected that inherently compromise the system’s security. These configuration issues are defined as follows:

- Errors - Issues that prevent configuration or encryption from finishing.
  - Miss Matched Configuration File
    - The configuration file’s name or hash does not match the binary file.
  - Miss Matched Cryptographic Support File
    - The cryptographic support file’s name or hash does not match the binary.
  - Missing PESIPS Module General Section
    - No general section of PESIPS module could be located.
  - Missing PESIPS Module Config Section
    - No configuration section of PESIPS module could be located.
  - Discontinuous PESIPS Module General Sections
    - General PESIPS module sections must be in continuous memory so the integrity kernel may verify them using only one hash.
    - This is error is ignored if there is not integrity kernel detected.
  - Parse Error
    - Configuration or Cryptographic Support or Binary parse error.
• **Warnings**

  * Missing PESIPS Module IK Section
    - No integrity kernel section of PESIPS module could be located.
    - This is not an error if the system does not require an integrity kernel as possibly the SK is implemented all in secure memory such as obfuscated ROM.

  * Irregular Entry Point
    - Entry point of binary is in irregular section that is not marked as “.pesips_general_” or “.pesips_ik_”.

  * Mismatched Naming Convention
    - A section’s name and assigned type mismatch.

  * Un-Verified Section
    - A “.pesips_general_” or SWIP “*” section are not being verified by a hash digest.
    - These sections must be stored in secure memory such as obfuscated ROM to avoid tampering.

  * Un-Encrypted Section
    - A SWIP “*” section is not marked for encryption.
    - These sections may be read if they contain sensitive information, they should be moved to the “.pesips_general_” section if they do not.

Additionally it is the developer’s responsibility to ensure that the memory locations that sections are targeted to for decryption are inside of the systems trust boundary. The IPE has no functionality to ensure that this occurs.

### 6.6 PESIPS and the Trusted Platform Module

Developers familiar with the Trusted Platform Module (TPM) standard proposed by the Trusted Computing Group will undoubtedly notice similar applications and functionality to PESIPS. However, the TPM standard is meant to act as a broad hardware based security solution for a great deal of security issues including complex digital rights management and e-commerce. PESIPS on the other hand is a primarily software solution directly aimed at securing SWIP on embedded systems. This is not to say that their purpose or methods are mutually exclusive as the TPM standard provides for a secure boot procedure, including system integrity verification. Similarly it provides mechanisms for storing keys used to encrypt blocks of data for purposes like key storage. However, the TPM standard is focused on hardware support to enable a broad range of these features as opposed to the actual implementation of a specific application like SWIP encryption. In this way PESIPS can function in a complimentary manor to the TPM standard where a TPM module can act as secure system storage or facilitate security kernel integrity verification at boot.
6.7 Summary

In this chapter we have:

- Highlighted the overlap between legacy and commodity SWIP security solutions.

- Proposed a Portable Embedded Software Intellectual Proper Security (PESIPS) to enable rapid, easily configurable deployment of SWIP security on a variety of platforms.

- Detailed the functionality of the PESIPS system including the IPE and PESIPS Module.

- Performed an overview of the requirements behind deploying and configuring the PESIPS system.
Chapter 7

Results

7.1 Overview

Through the course of this work we have sought to secure SWIP found on a wide range of systems. This effort has yielded three separate implementations to serve as a proof of concept for our legacy, commodity, and PESIPS solutions. In this chapter we present the result of these implementations including their demonstration systems, resource utilization, and performance metrics.

7.2 Legacy Embedded System Implementation

This section presents a basic proof of concept demonstration system for our legacy solution.

7.2.1 Demonstration System

The demonstrator hardware contains a Spectrum Digital C55 Development board (C55) featuring a TMS320C5509A Texas Instruments 32-bit DSP. Development of the security kernel and supporting software is accomplished under the Texas Instruments Code Composer Studio 3.1 (CCS). Implementation of the IPE and simulated key server is performed under cygwin 1.5.25 and GNU Compiler Collection 3.4.4. Once the application (containing the SK) is compiled by CCS it is encrypted on the PC by the IPE before deployment over JTAG into the Flash memory of the C55 board. The key used to encrypt the SWIP by the IPE is deployed into the simulated server. During operation the link between the C55 and key server is simulated through the use of a USB based data link. The DS2432 is simulated in the key sever and deployed to the C55 through a general purpose input/output connection (GPIO).
7.2.2 Software Utilization

The complete on-chip memory space of the C55 contains 256 Kilobyte, and we assume that the security kernel will never decrypt more than this amount of code during boot. The memory utilization of the entire solution is approximately 28.7Kb or 8.9% of the available onboard memory on the C55. As Table 7.1 illustrates that the AES block cipher account for 42.1% of this which can be attributed to its large lookup table. This provides an excellent design point where a more light weight implementation of AES or entirely different block cipher can be substituted to decrease size. By far the most code intensive section of our implementation is that of the Diffie-Hellman key exchange and its underling arithmetic support which occupies 33.3% of the implementation. On the other hand authentication has a minimal impact as the functionality is contained primarily in the hardware DS2432. The only code deployed to the C55 is simply to interface to the device over a GPIO port.

Table 7.1: Legacy System Solution Memory Utilization

<table>
<thead>
<tr>
<th>System Element</th>
<th>Components</th>
<th>Bytes</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Legacy Solution</td>
<td>Security Kernel</td>
<td>5259</td>
<td>17.9%</td>
</tr>
<tr>
<td></td>
<td>Authentication</td>
<td>1988</td>
<td>6.8%</td>
</tr>
<tr>
<td></td>
<td>Key Exchange</td>
<td>9781</td>
<td>33.3%</td>
</tr>
<tr>
<td></td>
<td>AES</td>
<td>12366</td>
<td>42.1%</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>29394</td>
<td></td>
</tr>
<tr>
<td>Key Exchange</td>
<td>Diffie-Hellman</td>
<td>1448</td>
<td>14.8%</td>
</tr>
<tr>
<td></td>
<td>Davies-Mayer Hash</td>
<td>672</td>
<td>6.9%</td>
</tr>
<tr>
<td></td>
<td>ECC</td>
<td>2474</td>
<td>25.3%</td>
</tr>
<tr>
<td></td>
<td>Modular Arithmetic</td>
<td>5187</td>
<td>53.0%</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>9781</td>
<td></td>
</tr>
<tr>
<td>Authentication</td>
<td>Authentication Protocol</td>
<td>305</td>
<td>15.3%</td>
</tr>
<tr>
<td></td>
<td>SHA1</td>
<td>1683</td>
<td>84.7%</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>1988</td>
<td></td>
</tr>
</tbody>
</table>

7.2.3 System Performance

Through testing on our demonstrator components we obtained an average performance of approximately 21 million cycles or 105 milliseconds for one DS2432 Authenticated ECDH exchange on the C55 processor. This value is obtained by performing several different key exchanges with different 256-bit scalar values. We then compared this performance with several different published implementations. The comparison is done in seconds normalized over the operating frequency of the platform. We also evaluated the symmetric-key encryption performance on the C55 and evaluated that to be 2023 cycles per 128 bits. We can also observe that the symmetric-key encryption speed is 3 orders of magnitude faster than
public-key encryption. For the complete protocol, we evaluated that the ECDH handshake and subsequent decryption of 128 Kbytes of firmware takes about 40 million cycles on the C55. Since ECDH consumes 20 million cycles, it thus takes roughly the same amount of time to decrypt a block of 128 Kilobytes of code as it takes to perform two ECC point multiplications (one complete ECDH handshake). Hence, we conclude that it would not make sense to optimize the current symmetric-key decryption speed without doing a corresponding optimization in the ECDH protocol implementation.

7.3 Commodity Embedded System Implementation

This section presents a basic proof of concept demonstration system for our design flow. We assume the binding of a single SWIP binary to an embedded system during a trusted production process. The underlying hardware architecture has been selected with this scenario in mind.

7.3.1 Demonstration System Hardware Architecture

Figure 8 shows the different components of the hardware architecture used for our prototype implementation on a Xilinx Spartan XC3S500E FPGA. A Microblaze soft core processor integrates several co-processors attached through a 32-bit processor local bus. The PUF is attached as a co-processor to the Microblaze using a dedicated fast simplex link (FSL). The block rams (BRAM) are not used as on-chip memory. Instead, the obfuscated ROM is used to achieve secure boot. The dotted box in 7.1 indicates the boundary of the FPGA. Anything outside it is an off-chip component, and is non-trusted.

Figure 7.1: SWIP binding prototype system architecture on a Xilinx Spartan XC3S500E FPGA.
7.3.2 Physical Unclonable Function (PUF)

For our prototype implementation, we used a ring-oscillator (RO) based PUF that has been proposed in [65] using several identical ROs. This PUF exploits random but static manufacturing process variations in RO frequencies. The PUF output is created by pairwise comparison of the RO frequencies. These comparisons can be represented as a challenge/response function, where the chosen ring oscillator pair is the challenge, and the comparison result is the response. An RO PUF has been selected due to its suitability for an FPGA implementation. A single RO circuit is created as a hard macro and instantiated several times to build the PUF as proposed in [1]. Enrollment of the PUF, the process of extracting the challenge/response pairs for the first time, is performed by simple C-program during the trusted development phase. The encryption keys, required for SWIP encryption, are derived from the PUF enrollment. PUF outputs are noisy by nature; a finite number (though low) of PUF output bits vary over time. However, for cryptographic operation, a stable key is necessary. Generating noise free keys from PUF is an active area of research, and several error correction schemes have been proposed such as [43]. Most of them require a complex implementation in software and/or hardware. The main objective in this work is to show the use of a PUF in the SWIP protection mechanism, and therefore, we have implemented a simple but effective error correction mechanism. By only selecting the RO pairs that have a relatively high difference in frequency we can extract a stable key and ignore error prone RO pairs. The drawback of this scheme is that we are not able to fully utilize all the ring oscillators for key extraction. However, other error correction techniques can be employed in a production scenario.

7.3.3 Functionality Testing

We have tested our prototype design on five Xilinx Spartan XC3S500E FPGAs. PUF enrollment was done on all of the FPGAs to extract their respective 128-bit keys and a single C-code binary was encrypted using each of these keys to produce five encrypted binaries. To validate the SWIP binding functionality, these five binaries were executed individually on each of the five sample FPGA chips. Each of the FPGAs could successfully execute exactly one binary with no two of them being able to execute the same binary.

7.3.4 PUF Characterization

We measure two parameters namely uniqueness and reliability to characterize the PUF. We define uniqueness as an estimate of how clearly a PUF can distinguish an FPGA from another. In other words, it estimates the difference between two keys generated by the PUF on two different FPGAs. Reliability expresses the stability of a specific response that is produced by a PUF from an FPGA for a given challenge. A stable PUF should reproduce the same response with minimum rate of error for a particular challenge if the challenge is applied to the PUF multiple times. Table 7.2 shows the metrics of the PUF that we
implemented.

Table 7.2: PUF Metrics

<table>
<thead>
<tr>
<th># of RO's</th>
<th>Uniqueness</th>
<th>Reliability</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>44%</td>
<td>96.7%</td>
</tr>
</tbody>
</table>

The PUF performs with a high reliability while giving a moderately high value of uniqueness. The reliability figure of 96.7% is calculated for all 255 pairs of ring oscillators during the PUF enrollment although a 100% reliable key is generated using the most stable 128 pairs. We provide these figures to verify the functionality of our proof of concept system and refer to Maiti et al. [43] for a more detailed evaluation of the reliability and stability ring-oscillator based PUFs.

7.3.5 Hardware Utilization

Table 7.3 shows the overall FPGA slice count used for the whole design is 3678, although the components specific to our design flow (i.e. the PUF and the ROM) only need 745 slices.

Table 7.3: Commodity Solution Resource Utilization Xilinx Spartan XC3500E

<table>
<thead>
<tr>
<th>Component</th>
<th>LUTs</th>
<th>Slices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microblaze</td>
<td>1397</td>
<td>698</td>
</tr>
<tr>
<td>PUF</td>
<td>931</td>
<td>279</td>
</tr>
<tr>
<td>Obfuscated ROM</td>
<td>559</td>
<td>279</td>
</tr>
<tr>
<td>DDR Ram Controller</td>
<td>1304</td>
<td>1024</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>4563</strong></td>
<td><strong>3678</strong></td>
</tr>
</tbody>
</table>

7.3.6 Software Utilization

As evident from the Table 7.4, the software memory overhead required for our particular implementation can be considered sizeable for the overall internal memory of a Spartan XC3S500E FPGA, over 50% of 32KB available. However, this implementation is a proof of concept. Our design flow is flexible to allow different hash and decryption primitives to be utilized to achieve smaller code size, higher security or faster execution time. Simply replacing AES with XXTEA for general purpose encryption reduces the over footprint of our system to 20.1% or about 6.7 KB.
Table 7.4: Commodity System Solution Memory Utilization

<table>
<thead>
<tr>
<th>System Element</th>
<th>Components</th>
<th>Bytes</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commodity Solution</td>
<td>Integrity Kernel</td>
<td>196</td>
<td>1.1%</td>
</tr>
<tr>
<td></td>
<td>Security Kernel</td>
<td>6324</td>
<td>37.0%</td>
</tr>
<tr>
<td></td>
<td>AES</td>
<td>10560</td>
<td>61.8%</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>17080</td>
<td></td>
</tr>
<tr>
<td>Integrity Kernel</td>
<td>Kernel</td>
<td>28</td>
<td>14.3%</td>
</tr>
<tr>
<td></td>
<td>Boot/Interrupt Vectors</td>
<td>80</td>
<td>40.8%</td>
</tr>
<tr>
<td></td>
<td>XXTEA</td>
<td>80</td>
<td>40.8%</td>
</tr>
<tr>
<td></td>
<td>Expected Hash</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Results</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>196</td>
<td></td>
</tr>
<tr>
<td>Security Kernel</td>
<td>Kernel</td>
<td>4160</td>
<td>65.8%</td>
</tr>
<tr>
<td></td>
<td>SHA1</td>
<td>1908</td>
<td>30.2%</td>
</tr>
<tr>
<td></td>
<td>Section Info (64 Each)*4</td>
<td>256</td>
<td>4.0%</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>6324</td>
<td></td>
</tr>
</tbody>
</table>
7.3.7 System Performance

In our legacy system the computationally intensive task was performing the DS2432 authenticated ECDH key exchange protocol. Our commodity system removes the need for the use of ECC which accounted for the majority of this overhead. However, querying the PUF to extract a key also introduces a computational overhead. Enrollment requires 90 seconds to generate a stable key on average but is a onetime operation during development. On the other hand the time required for runtime key extraction is 4 seconds.

7.4 PESIPS Implementation

The Portable Embedded Software Intellectual Property Security system is designed to be readily customizable and deployable on a wide array of system. As such determining overhead and performance metrics are determined by various design solutions. For the purposes of completeness we present the utilization for PESIPS deployed on the same platform as our Commodity system.

7.4.1 Software Utilization

Table 7.5 shows the utilization of the PESIPS system as deployed on the Commodity system, utilizing the PUF as the system storage, using AES as the block cipher, and using SHA1 as the hash engine. Again we see rather high utilization for the Spartan XC3S500E FPGA, 65.8% or 21.1 KB of its available 32KB. This increase can be attributed to the added functionality to better suit a wider array of platforms and the root key functionality. Again a major source of overhead is the selection of AES as the block cipher. If XXTea was deployed in its place and reused for the hash engine through its application in the Davies-Mayer Hash then this could lower PESIPS footprint to 28.3% of available Spartan XC3S500E FPGA memory, or 9.1KB. We find that this is an excitable utilization for the added security that PESIPS offers. However, with further optimization this number could be lowered even further.

7.5 Summary

In this chapter we have:

- Presented proof of concept demonstration systems for our legacy, commodity, and PESIPS systems.
- Provided metrics highlight the implementation cost and performance of our solutions.
- Discussed the flexibility provided by substituting system components of our commodity and PESIPS systems.
Table 7.5: PESIPS System Solution Memory Utilization

<table>
<thead>
<tr>
<th>System Element</th>
<th>Components</th>
<th>Bytes</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Integrity Kernel</td>
<td>164</td>
<td>0.6%</td>
</tr>
<tr>
<td></td>
<td>Security Kernel</td>
<td>6496</td>
<td>23.3%</td>
</tr>
<tr>
<td></td>
<td>Component Management</td>
<td>500</td>
<td>1.8%</td>
</tr>
<tr>
<td></td>
<td>Configuration Storage</td>
<td>728</td>
<td>2.6%</td>
</tr>
<tr>
<td></td>
<td>Cryptographic Library</td>
<td>20008</td>
<td>71.7%</td>
</tr>
<tr>
<td></td>
<td><strong>Total</strong></td>
<td><strong>27896</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Kernel</td>
<td>76</td>
<td>38.8%</td>
</tr>
<tr>
<td></td>
<td>XXTEA</td>
<td>80</td>
<td>40.8%</td>
</tr>
<tr>
<td></td>
<td>Expected Hash Results</td>
<td>8</td>
<td>4.1%</td>
</tr>
<tr>
<td></td>
<td><strong>Total</strong></td>
<td><strong>164</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>System Storage Interface</td>
<td>468</td>
<td>64.3%</td>
</tr>
<tr>
<td></td>
<td>Root Storage Information</td>
<td>52</td>
<td>7.1%</td>
</tr>
<tr>
<td></td>
<td>Section Info (52 each) * 4</td>
<td>208</td>
<td>28.6%</td>
</tr>
<tr>
<td></td>
<td><strong>Total</strong></td>
<td><strong>728</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Management</td>
<td>1224</td>
<td>6.1%</td>
</tr>
<tr>
<td></td>
<td>Block Cipher (AES)</td>
<td>16876</td>
<td>84.3%</td>
</tr>
<tr>
<td></td>
<td>Hash Engine (SHA1)</td>
<td>1908</td>
<td>9.5%</td>
</tr>
<tr>
<td></td>
<td><strong>Total</strong></td>
<td><strong>20008</strong></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 8

Conclusion

The proliferation of embedded systems into nearly every aspect of modern infrastructure and society has seen their deployment in such diverse roles as monitoring the power grid and processing commercial payments. Software intellectual property (SWIP) is a critical component of these increasingly complex systems and represents a significant investment to its developers. However, deeply immersed in their environment, embedded systems are difficult to secure. As a result, developers want to ensure that their SWIP is protected from being reverse engineered or stolen by unauthorized parties. Many techniques have been proposed to address the issue of SWIP protection for embedded systems. These range from secure memory components to complete shifts in processor architectures. While powerful, these approaches often require the development of systems from the ground up or the application of specialized and often expensive hardware components. As a result they are poorly suited to address the security concerns of legacy embedded systems or systems based on commodity components.

8.1 Contributions

This work has presented three distinct efforts to address the issue of software intellectual property security on embedded systems. Our solutions have specifically targeted legacy and commodity based systems where specialized secure cryptographic components are available. As such we have explored software-centric solutions that employ SWIP encryption in combination with secure key storage and authentication solutions. We focus on SWIP encryption to address one of the larger issues with SWIP security on such systems; it’s storage in non-volatile memory. In our legacy system we take advantage of the communication feature found on many legacy systems such as embedded sensor nodes. By encrypting SWIP then storing the key remotely we ensure that an attacker can not readily copy SWIP stored in non-volatile system Flash. This process is made safe through the application of a key exchange protocol to establish a confidential communication channel. However, the lack of authentication is a significant issue for such an approach and requires the introduction of a secure component, the DS2432. Leveraging this component we develop an authenticated key exchange protocol to create a confidential and authenticated communication channel. One of the major limitations of our legacy solution was the ability for an attacker to observe system communication over buses and exploit this to attack our solution. In our FPGA based
commodity system we were able to address this issue by exploiting a physical unclonable function to extract a unique identifier (to use as a key) from the physical characteristics of the FPGA device. Coupled with the system on chip nature of FPGA based designs we were able to provide a robust and simple SWIP security system. Based on our initial work in the legacy and commodity systems our work has culminated with the presentation of a generic Portable Embedded Software Intellectual Property Security (PESIPS) system. PESIPS offers a flexible and easily configurable platform for the deployment of the core ideas used in both commodity and legacy systems. When paired with a secure storage mechanism such as PUF key storage on our FPGA based commodity system we have a system that can:

- Guarantee the confidentiality of SWIP to prevent its theft.
- Insure the integrity of system software to prevent malicious modification.
- Limit the use of SWIP only to authentic customers and system.
- Ensure that authenticity of SWIP running on a system.

Coupled with a flash based FPGA device and a secure storage mechanism such as PUF, the PESIPS system can serve as a stepping stone for developing future embedded system with robust features to prevent SWIP theft.

8.2 Future Work

Several avenues exist for continued work, specifically on expansion of the PESIPS system including:

- Relocation support to enable swapping of multiple encrypted sections at runtime allowing the use of protected binaries that are larger than trusted internal memory.
- Formal support for the use of external RAM through a generic runtime encryption scheme.
- Expansion of system flexibility by supporting more than one concurrent hash or block cipher.
- Support for extended cryptographic features such as digital signatures.
- Extension of system features to include common tasks like remote software update.
- Modularization to allow selection of system components to allow for easier customization.
Bibliography


