A COMPREHENSIVE SECURITY ANALYSIS OF AND AN IMPLEMENTATION FRAMEWORK FOR EMBEDDED SOFTWARE ATTESTATION METHODS LEVERAGING FPGA-BASED SYSTEM-ON-A-CHIP ARCHITECTURES

THESIS

Patrick A. Reber, Second Lieutenant, USAF

AFIT-ENG-MS-17-M-063

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-17-M-063

A COMPREHENSIVE SECURITY ANALYSIS OF AND AN IMPLEMENTATION FRAMEWORK FOR EMBEDDED SOFTWARE ATTESTATION METHODS LEVERAGING FPGA-BASED SYSTEM-ON-A-CHIP ARCHITECTURES

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Electrical Engineering

Patrick A. Reber, BS

Second Lieutenant, USAF

March 2017

AFIT-ENG-MS-17-M-063

A COMPREHENSIVE SECURITY ANALYSIS OF AND AN IMPLEMENTATION FRAMEWORK FOR EMBEDDED SOFTWARE ATTESTATION METHODS LEVERAGING FPGA-BASED SYSTEM-ON-A-CHIP ARCHITECTURES

THESIS

Patrick A. Reber, BS
Second Lieutenant, USAF

Committee Membership:

Scott R. Graham, Ph.D.
Chair

Barry E. Mullins, Ph.D., P.E.
Member

Michael A. Temple, Ph.D.
Member

Abstract

As embedded devices continue to proliferate, the safety and security of critical applications increasingly relies on trust in the running firmware. Malicious actors who compromise these devices may alter this code to accomplish their objectives while remaining undetected. Integrity checks of this code alerts users to changes which indicate a potential attack. This is commonly referred to as software attestation, and utilizes a challenge-response protocol that allows a trusted verifier to integrity check the memory of an untrusted device.

This research assesses the potential for utilizing FPGA System-on-a-Chip (SoC) architectures for software attestation and analyzes the resulting security. A framework is proposed to leverage the SoC capabilities to quickly and easily perform software attestation with minimal runtime impact. The SoC hardware acts as a trusted local entity which aids in verification of the firmware run by the processor, while the incorporation of the SoC requires little processor software or configuration changes. This allows designs which previously utilized solely microcontrollers to easily port to this architecture for increased software security.

This framework is constructed with an example algorithm and confirmed to detect attacks. The resulting degradation in processor speed is examined, as well as the potential attestation speed. Processor degradation ranges from 0.5 to 42 percent and read speed ranges from 0.06 to 3.2GB/s. Multiple additional alternatives are implemented and analyzed, which increase security at the cost of speed and simplicity. Various potential options are characterized in a novel attestation taxonomy extension for comparison.

Acknowledgments

The completion of this work would not have been possible without the aid of many dedicated researchers along the way. I would like to first thank my advisor, Dr. Scott Graham, for his advice and guidance as I worked through every stage of research. His willingness to devote the time and energy required to develop both myself as a researcher and this work as a valuable product contributed significantly to the completion of this thesis. I would also like to thank my sponsors at AFRL for supporting my research, as well as producing ideas which pique my interest as well as contribute to real world applications.

Patrick A. Reber

Table of Contents

List of Figures

List of Tables

List of Acronyms

AES	Advanced Encryption Standard
ARM	Advanced RISC Machines
AXI	Advanced eXtensible Interface
BBRAM	Battery-Backed RAM
ECB	Electronic Code Book
ECU	Electronic Control Unit
FPGA	Field Programmable Gate Array
FSBL	First Stage Boot Loader
GPU	Graphics Processing Unit
GSRD	Golden System Reference Design
HMAC	Hash Message Authentication Code
HPS	Hard Processor System
HSM	Hardware Security Module
LUT	Look-Up Table
MPU	Microprocessor Unit
MUX	Multiplexer
NVM	Non-Volatile Memory
PLB	Programmable Logic Block
PLC	Programmable Logic Controller
RISC	Reduced Instruction Set Computing
RNG	Random Number Generator

ROP	Return-Oriented Programming
SDRAM	Synchronous Dynamic Random-Access Memory
SEU	Single-Event Upset
SRAM	Static Random-Access Memory
SoC	System on a Chip
SWAP	Size, Weight, and Power
TPM	Trusted Platform Module
VHDL	VHSIC Hardware Description Language
WSN	Wireless Sensor Networks

A COMPREHENSIVE SECURITY ANALYSIS OF AND AN ANIMPLEMENTATION FRAMEWORK FOR EMBEDDED SOFTWARE ATTESTATION METHODS LEVERAGING FPGA-BASED SYSTEM-ON-A-CHIP ARCHITECTURES

I. Introduction

1.1. Background and Motivation

Embedded devices numbering in the tens of billions enable markets including automotive, aviation, communications, entertainment, industrial, medical, automotive, and military [1]. Now permeating every aspect of modern society, security in networking and remote access applications remains a concern [2], with devices being compromised by malware at every level of complexity [3], [4]. Increasing this risk, firmware designed for embedded systems often utilizes common libraries, which contain flaws exposing many devices to the same vulnerability [5]. These vulnerabilities allow attackers to gain control of the devices and insert malicious functionality [6], potentially harming individuals, businesses and governments alike [7].

Remote attestation aims to reduce this threat through detection of compromise, by enabling an inspected device to make a claim about its properties to a verifier over a network. The verifier then decides whether or not to trust the device [7]. Commonly referred to as software attestation, verification of executable code detects any modifications to the instruction memory of an application [8]. Utilizing attestation can defend against common exploits including buffer overflows and code injection attacks [9], [10].

Proposed attestation solutions range in complexity from relatively simple memory checksums [8] to reliance on trusted hardware platforms [10] or reconfigurable hardware solutions [11] to full virtualization architectures [7]. This research focuses on the

implementation of remote attestation in trusted environments utilizing FPGA system-on-a-chip (SoC) architectures, which include a microcontroller and FPGA fabric on one chip.

1.2. Problem Statement

This research presents the design, development, and characterization of a framework for the utilization of FPGA SoC architectures to implement dynamic hardware-based software attestation on embedded devices. As SoCs increase in popularity and grow in capability, these devices present a unique opportunity to incorporate security checks into embedded applications which typically lack any form of attestation.

1.3. Research Objectives

The goal of this research is to assess the feasibility of utilizing emerging FPGA SoC architectures for software attestation. The hypothesis is that SoCs allow for a feasible attestation framework to provide additional attestation capabilities and speed not previously possible in software-based attestation schemes, while reducing processing impact. This research intends to create a system which allows designers to utilize SoCs in systems which previously used only microcontrollers, enabling them to retain the same functionality, while simultaneously incorporating dynamic software attestation and continuing to meet hard real-time requirements.

1.4. Organization

Chapter II provides background relevant to the understanding of this research. Security concepts are discussed, along with various attestation architectures and previous research limitations. Chapter III discusses a use case and threat model. Relevant assumptions are

presented along with design decisions and methodology used to construct an attestation framework. In Chapter IV, experiments are proposed to characterize the effect of the proposed framework are discussed. Chapter V details the results of the experiments conducted and Chapter VI presents threat mitigation and analyzes the framework solution proposed. Chapter VII concludes with a summary and a discussion of future objectives for continuing research.

II. Background

This chapter provides background information and context related to the design of a system utilizing SoCs for security. It summarizes relevant technologies leveraged for the solution, as well as the specific hardware architecture used. This chapter also describes research in the fields of both software attestation methodologies as well as SoC security.

2.1. Field Programmable Gate Arrays

First introduced in 1984, FPGAs are comprised of programmable logic blocks (PLB) and interconnects [12]. PLBs commonly contain digital circuits including Look-Up Tables (LUT), multiplexers (MUX), and flip-flops. Figure 1 shows a generic FPGA architecture with square PLBs and circular interconnects. A file, generally referred to as a 'bitstream' configures these blocks and interconnects them to construct circuits. These circuits are defined through code written in hardware languages including VHDL or Verilog, or transpiled to hardware languages from a higher-level language such as C or Python.

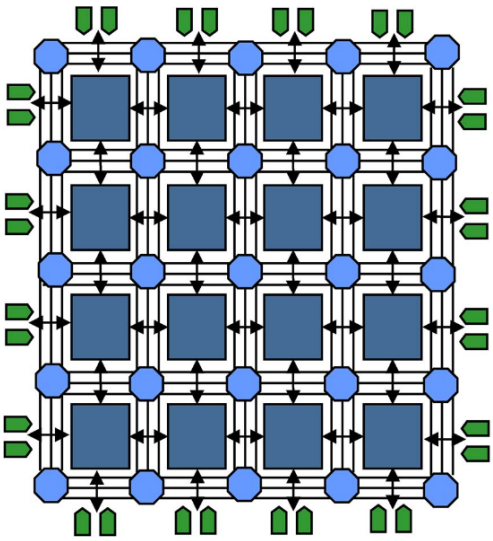

Figure 1: Generic FPGA Architecture [12]

As technology improves, manufacturers are able to pack more features into each PLB, and pack more PLBs onto each FPGA. In addition, cost and power have decreased, while speed has increased. One example of technology increase is shown in Figure 2, mapping the improvement in Xilinx FPGAs over two decades. These trends led to the increase in the use of FPGAs. The largest FPGA manufacturer, Xilinx, grew from $100 million in sales in 1991 to $2.2 billion in 2013.

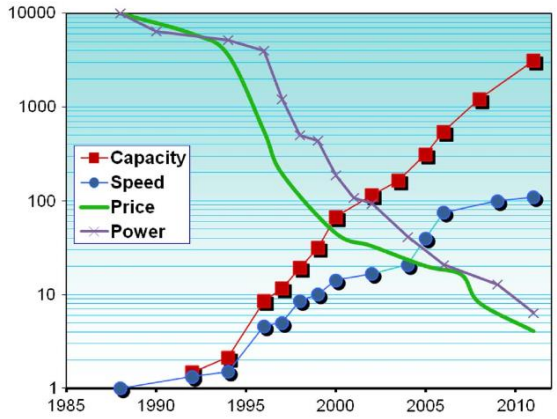

Figure 2: Xilinx FPGA Capability Over Time [12]

FPGA chips belong to a few basic categories of technology: SRAM, Flash, and Antifuse [13]. SRAM FPGAs use internal volatile static latch cells to store configuration data [14]. After power-on, they transfer the bitstream from an external non-volatile memory source to the FPGA for configuration. Flash FPGAs use internal non-volatile memory to store the configuration data. This allows for faster power-on, no external bitstream transfer, and information transfer through power-cycles [14]. Antifuse FPGAs program through a pulse, which irreversibly connects the hardware, making them one time programmable [14].

As SRAM FPGAs represent the most common type of chip, the majority of security research centers around these chips. The research conducted in large part aims to reduce the risk to manufacturers who invest into the development of FPGA designs for consumer goods [15]. For example, the most common security vulnerability of SRAM FPGAs allows for attackers to

intercept the bitstream as the FPGA is programmed, and subsequently alter the programming, perhaps including additional and unauthorized functionality [15].

2.1.1. System on a Chip

System on a Chip is a general term used to describe architectures which include multiple components contained on a single chip. For example, the Apple A7 SoC contains an ARMv8-A microprocessor, a Graphics Processing Unit (GPU), an image processor, and multiple caches. FPGA manufacturers including Altera, Xilinx, and Microsemi refer to SoCs as their products which contain FPGA fabric as well as a microcontroller unit. In the context of this research, SoC refers to a chip which contains at least an FPGA and a microcontroller, though they may contain significant additional features. SoC architectures have grown to include multiple processor cores, graphics processing units, memories, and security blocks, among other features. Figure 3 shows one high-end example, the Zynq UltraScale+ SoC from Xilinx, which includes multiple processors, a GPU, dedicated security modules, and additional performance components.

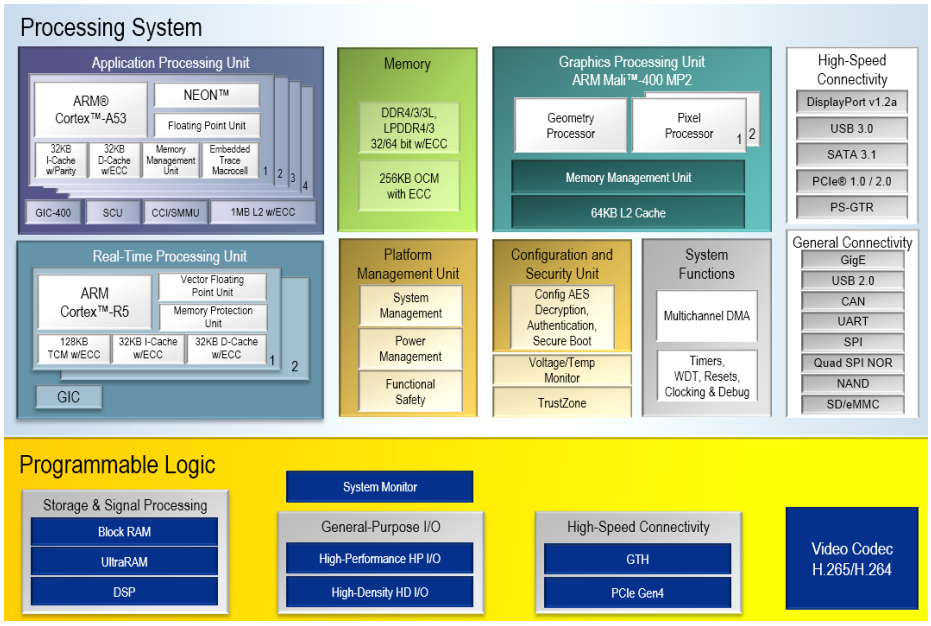

Figure 3: Example Xilinx SoC Architecture [16]

2.1.2. Cyclone V SoC

This research uses a Cyclone V ("Cyclone Five") chip from Altera. Of the six variants available, three contain fully featured ARM Cortex-A9 dual-core processors, as well as required embedded peripheral components for a fully featured microcontroller. Altera refers to the microcontroller in the Cyclone V as the Hard Processor System (HPS).

2.1.2.1. Architecture

The Cyclone V FPGA fabric contains PLBs and configurable logic, as well as blocks for transceivers, memory controllers, configuration and encryption and clock logic. The HPS contains the ARM microprocessor and a suite of peripheral components. Both the FPGA and the HPS contain interface logic which allows communication across domains. Figure 4 shows a layout of a Cyclone V chip with FPGA components on the right, HPS components on the left, and interfaces between the two sides.

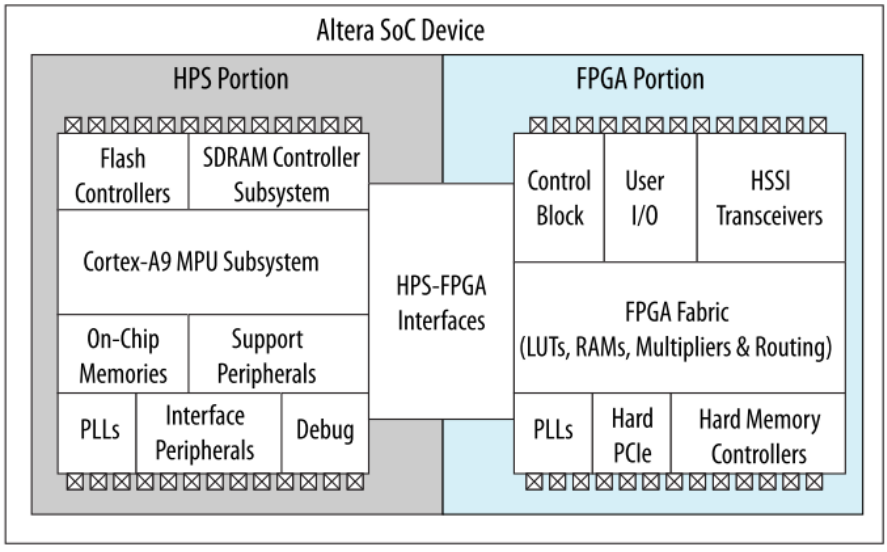

Figure 4: Cyclone V Architecture with HPS [17]

2.1.2.2. Communication Bridges

In the Cyclone V, communication between the FPGA and the HPS occurs through the "HPS-FPGA Interfaces" shown above. The interfaces are composed of three main bridges, shown in the figure below. These are the FPGA to HPS bridge, the HPS to FPGA bridge, and the Lightweight HPS to FPGA bridge. These bridges communicate with configurable width AXI or Avalon interfaces [18]. These bridges are enabled through preloader configuration and require functional logic in both the HPS and the FPGA for communication. In addition to these three bridges, there are channels for configuration data between the FPGA and the HPS, as well as a bridge for the FPGA to master memory accesses to the HPS SDRAM controller. Off chip memory is accessed through the memory controller, stores executable code and variables, and supports up to 4GB of addressable memory. Figure 5 shows the block diagram of this architecture, along with peripherals located in the Cyclone V HPS.

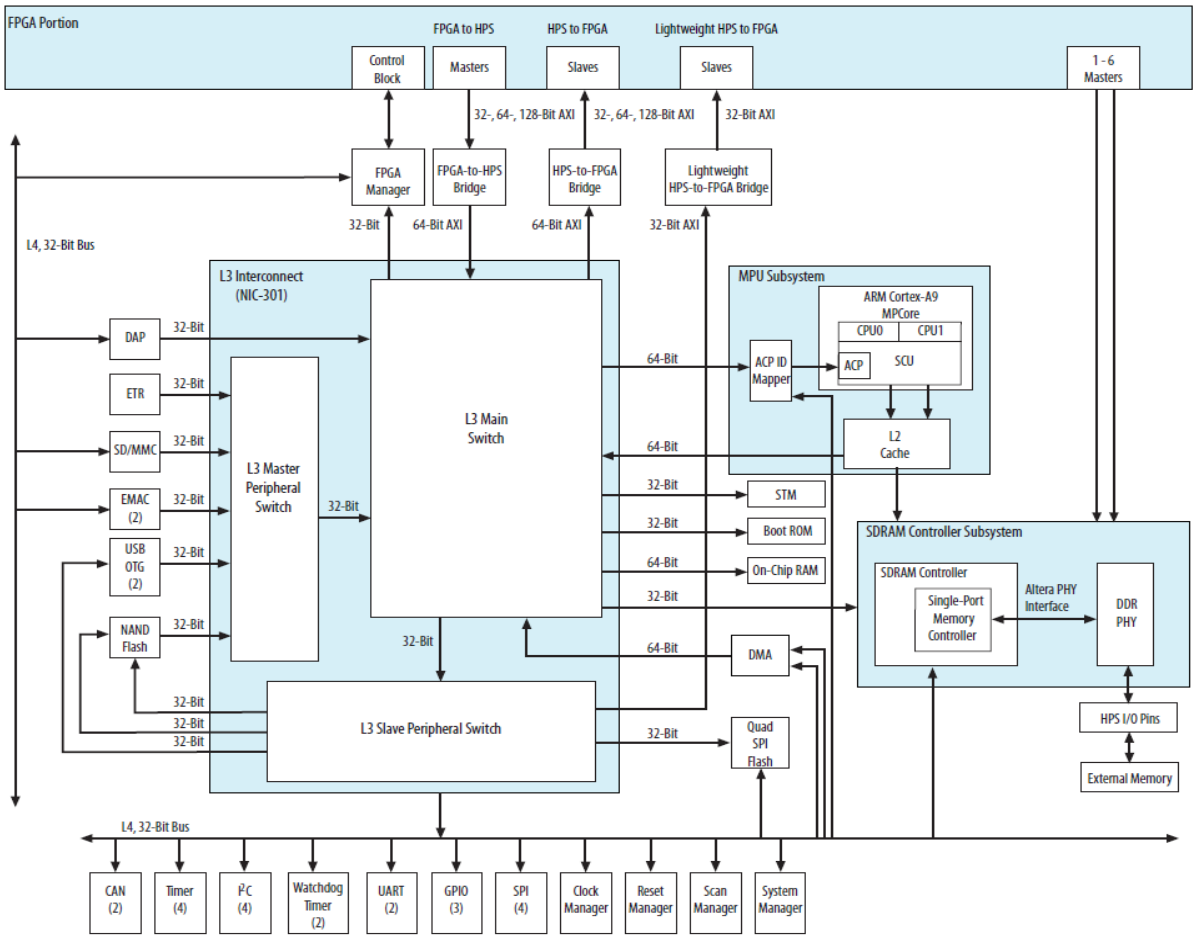

Figure 5: Cyclone V HPS Architecture [17]

2.1.2.3. Boot Options

The Cyclone V SoC may be configured in several ways, and an initial design decision of setting pin values to create a boot procedure affects the final product in factors including flexibility, security, and configuration time [19]. In the cases shown, the connections between the HPS and the FPGA are configured in the preloader and the bitstream, respectively. In addition, any reprogramming must originate from the same source as the original boot. Figure 6, Figure 7, and Figure 9 show the three options for booting the device. In all cases, Figure 8 shows a typical boot procedure for the HPS. The FPGA only requires a single configuration step

with a bitstream, which may be encrypted. The boot mode is chosen between separate, HPS first, and FPGA first.

- HPS and FPGA Boot Separately

Figure 6 shows the first option for configuration of the SoC, where the FPGA and the HPS both obtain their required configuration information from independent external non-volatile memory (NVM) sources.

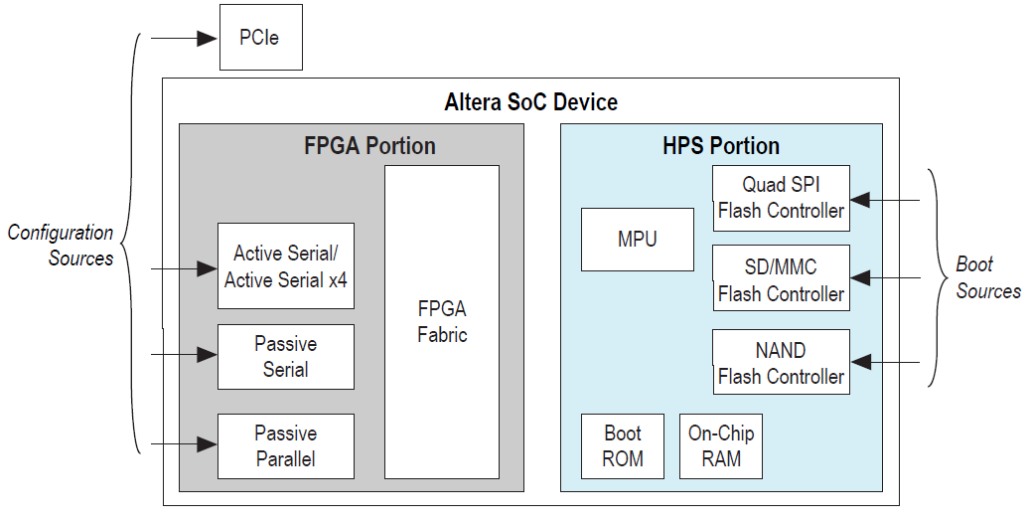

Figure 6: Altera SoC Separate Boot [20]

- HPS Boots First

Figure 7 shows the next option for configuration of the SoC, where the HPS configures first and subsequently programs the FPGA. In this option, the HPS may configure the FPGA at any point during the boot steps shown in Figure 8. Every stage after the boot ROM runs is user modifiable, and designers may choose to alter the steps shown in Figure 5 to load a bare-metal application instead of a full operating system.

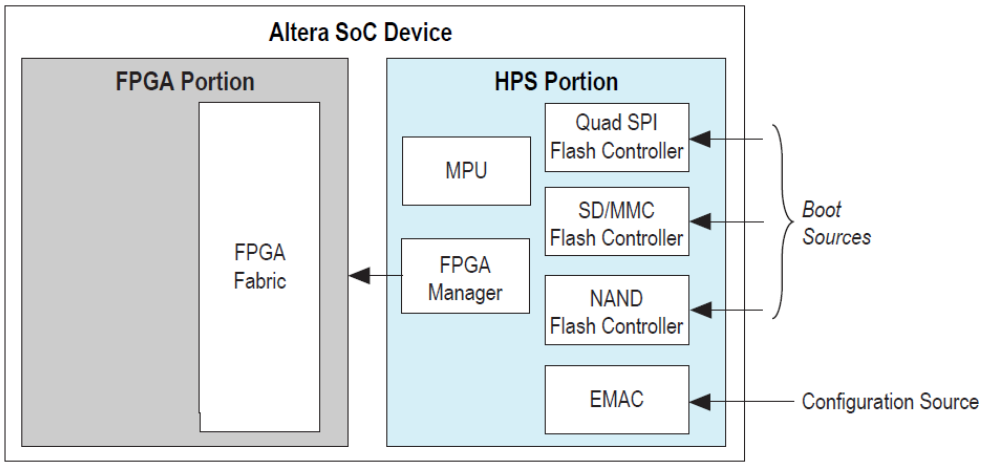

Figure 7: Altera SoC HPS Boot [20]

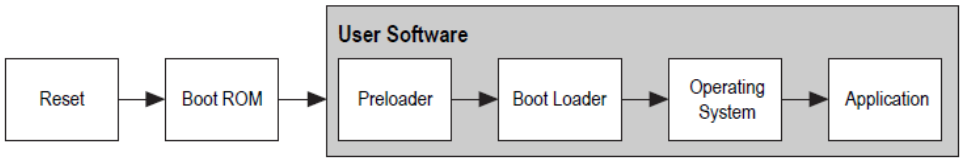

Figure 8: Altera SoC HPS Boot Flow [20]

- FPGA Boots First

Figure 9 shows the last option for configuration of the SoC, where the FPGA configures first, and stores HPS files behind a memory-mapped bridge for the processor to transfer and execute. The FPGA stores a preloader, and optionally a bootloader or bare metal application.

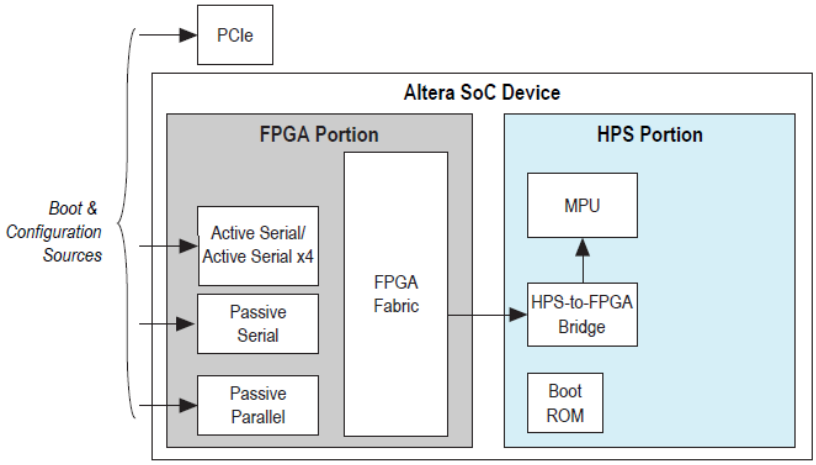

Figure 9: Altera SoC FPGA Boot [20]

2.2. SoC Security

As SoCs move into more critical fields, security requirements increase. Along with motivation to protect IP in applications, SoC security has become an area of significant concern for manufacturers. These security capabilities can be broken down by goal into the typical information assurance tenets of confidentiality, integrity, and availability. Due to the typical consumer nature of FPGAs and the costly development of hardware designs, confidentiality has been the primary concern for manufacturers and researchers. The following capabilities exist in various SoC chips available from the two largest SRAM FPGA manufacturers: Xilinx and Altera.

2.2.1. Confidentiality

In order to protect the confidentiality of the FPGA configuration data, encryption and authentication are offered, using AES encryption and decryption and a hash message authentication code (HMAC) [21]. The keys used for the AES and hash functions are stored in either volatile battery-backed RAM (BBRAM) or non-volatile eFUSE. The choice between these two options depends upon application priorities. BBRAM keys allow for increased tamper resistance and passive clearing (addressed below). BBRAM also requires an external battery, which presents increased maintenance concerns. Alternatively, eFUSE requires no battery, but does not support key reprogrammability or passive clearing.

Passive clearing available in the BBRAM key mode allows for additional confidentiality through key and memory clearing. On detection of a tamper event, the design, including the encryption keys, can be erased through an intentional removal of battery power [22].

2.2.2. Integrity

Integrity for FPGA designs is primarily addressed through encryption and authentication of configuration bitstreams. This helps defeat reverse engineering and bitstream modification attacks [15], [23]. Processor integrity is primarily addressed through secure boot technology. The chain of trust begins with the boot ROM, which authenticates the first stage boot loader (FSBL), which authenticates the next stage, and so on until the final load takes place. Figure 10 shows the boot procedure for a Xilinx SoC, where the Bitstream may also be encrypted and authenticated in the FPGA as described above. Once the OS boot completes, the user retains responsibility for ensuring integrity of apps and appropriately restricting access to apps and FPGA configuration logic [24].

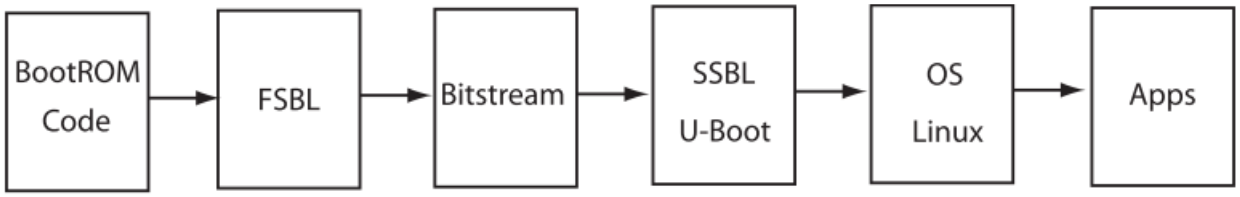

Figure 10: Secure Boot of Xilinx SoC [24]

2.2.3. Availability

Security research related to availability while using SoC architectures is somewhat limited. Research does exist on the prevention on single event upsets (SEU) which may alter program execution [25]. There are also relevant design documents describing remote update procedures which allow for graceful failure while retrieving a known safe state [26].

2.2.4. Proposed Uses

As a result of the security capabilities described above, SoC architectures have been recommended for applications including Wireless Sensor Networks (WSN), automotive, cloud computing, and PLCs.

2.2.4.1. Wireless Sensor Networks

Wireless sensor networks rely on spatially disparate nodes operating independently to monitor conditions and report to gateways. Due to the lack of physical security often present in WSN nodes, hardware which natively provides trust is ideal. SoCs allow for the addition of trusted hardware through FPGA-stored immutable security keys used to boot and communicate with the device [27]. In addition, SoCs have been proposed to be utilized to perform distributed on-demand software integrity checks to prevent the spread of malware between nodes [28].

2.2.4.2. Automotive

Research ventures such as the Evita Project [29] propose hardware security modules (HSM) to incorporate security primitives including hardware accelerated encryption, hashing, random number generators (RNG), tamper protection, internal non-volatile memory, and internal CPUs. The HSM would be used to create secure communication channels between UCEs, manage keys, and update firmware [30]–[32]. The proposed architecture with all required and optional modules is shown in Figure 11. SoCs allow for the capability to fill every requirement of the proposed HSM. The first implementation and analysis of the HSM shown was implemented on a Xilinx SoC [32]. A combination of the HSM implementation, combined with

software attestation at each node may present a unique application to utilize SoCs for automotive security [33].

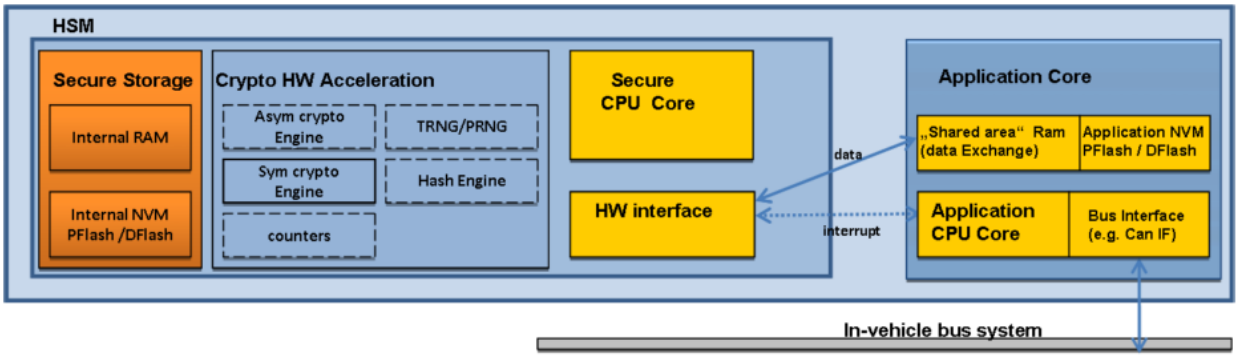

Figure 11: Proposed HSM Architecture [31]

2.2.4.3. Cloud Computing

Due to their flexibility, reprogrammability, and confidentiality capabilities, SoC architectures allow for secure storage and computation on data. Acting as a simulated homomorphic encryption engine, SoCs can secure client data in untrusted physical locations [34]. In addition, the capability of reprogrammability through the resident processor enables user-friendly integration into cloud architectures [35].

2.2.4.4. Programmable Logic Control

Described in [36], industry 4.0 refers to the fourth industrial revolution through the use of cyber-physical systems, and the internet of things and services. Challenges currently faced by PLC designers to prepare for industry 4.0 systems include implementation of high-performance control, secure communication, connectivity, and interoperability [37]. FPGA SoC systems allow for the addressing of each of these challenges through hardware acceleration, cryptographic modules, and reprogrammability [38] [39].

2.3. Embedded Software Security

Embedded devices continue to grow in use in consumer, industrial, medical, auto, and military use. In addition, embedded software often contains multiple common vulnerabilities [5]. The threat is exacerbated by the increasing education effort to teach skills to hack embedded devices [6] and the common use of default configurations and passwords [3]. Even systems which were previously believed to be secure from such simple threats face increasingly sophisticated malware [4], [40]. Architectural solutions are needed which protect embedded devices from these threats and report compromise.

2.4. Software Attestation

Software attestation aims to provide a means for an untrusted device under test to prove to a remote trusted verifier that the code running has not been modified. Figure 12 shows a generic attestation protocol. The verifier sends a request to the device (or prover). The device then runs a local algorithm to compute a value, which it sends as a response. Based on the value and timing of the response, the verifier decides if the device memory was altered. An incorrect or delayed response indicates malicious activity on the device.

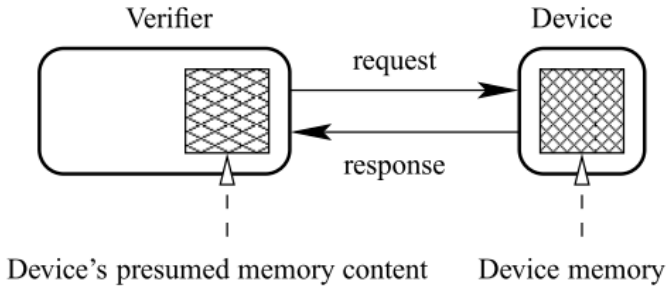

Figure 12: Generic Software Attestation Diagram [8]

Although software attestation potentially suffers from multiple attacks [41], a significant benefit to the use of software attestation is the fact that it performs attestation without secrets, a fact leveraged in key establishment and system recovery [42]. Various algorithms have been presented, varying from hardware, to software, to hybrid or FPGA-based approaches [8], [43]–[47]. The verifier represents the trusted entity in the system, while the device resides under full control of an attacker who may modify executing code. Common assumptions include trusted golden copy of memory (located at the verifier), that the attacker does not make hardware modifications to the device, and full optimization of the algorithm [48]. Common principles of software attestation include temporally fresh, comprehensive information, constrained disclosure, semantic explicitness, and trustworthiness, described in [7].

An excellent survey and taxonomy of attestation techniques through 2016 is presented in [48]. Figure 13 shows the graphical representation of this taxonomy. Table 1 shows this taxonomy, applied to attestation articles cited in this thesis, sorted by publication date. Refer to [48] for full descriptions of the categories.

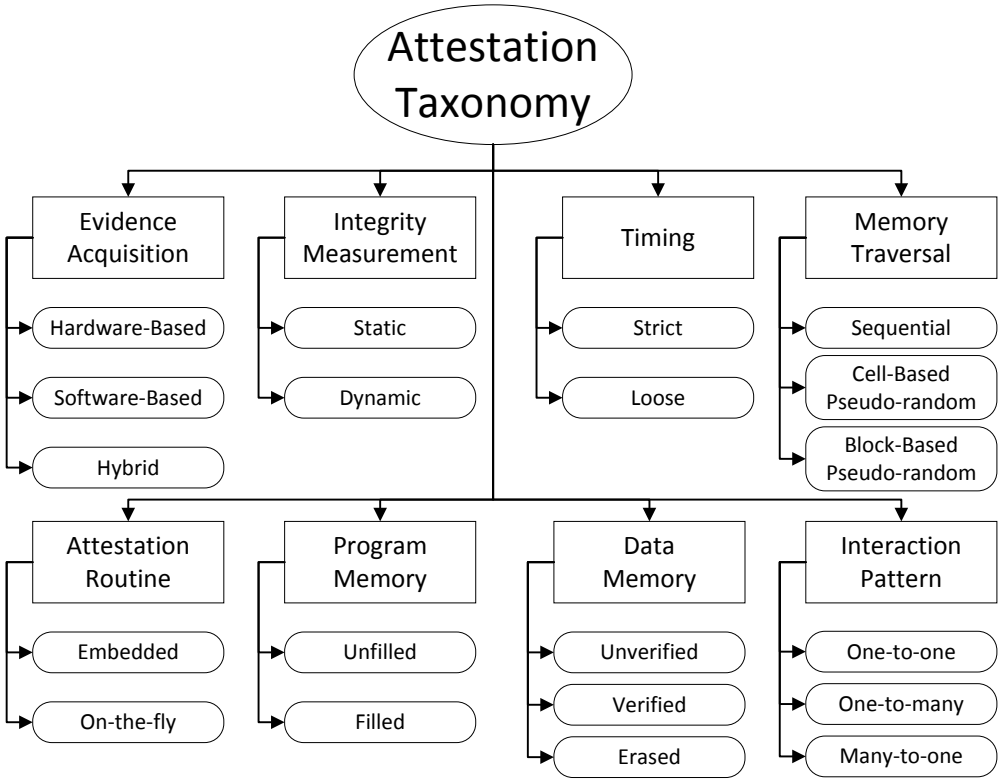

Figure 13: Attestation Taxonomy [48]

In the following table, CB refers to cell-based, BB refers to Block-Based, and PR refers to Pseudo-Random. The following sections expand on software attestation techniques organized by evidence acquisition methods: software, hardware, or hybrid.

Table 1: Cited Attestation Literature Organized by Taxonomy [48]

Approach	Evidence Acquisition	Integrity Measurement	Timing	Memory Traversal	Attestation Routine	Program Memory	Data Memory	Interation Pattern
Reflection (2000)	Software	Static	Loose	Sequential	Embedded	Unfilled	Unverified	One-to-one
SWATT (2004)	Software	Static	Strict	CB-PR	Embedded	Unfilled	Unverified	One-to-one
ICE (2004)	Hybrid	Dynamic	Strict	CB-PR	Embedded	Unfilled	Unverified	One-to-one
PIV (2005)	Software	Dynamic	Loose	Sequential	On-the-fly	Unfilled	Erased	One-to-one
Pioneer (2005)	Hybrid	Static	Strict	CB-PR	Embedded	Unfilled	Unverified	One-to-one
SAFE-OPS (2005)	Hybrid	Dynamic	Loose	N/A	Embedded	Unfilled	Unverified	One-to-one
CODESSEAL (2005)	Hybrid	Dynamic	Loose	N/A	Embedded	Unfilled	Unverified	One-to-one
M-TREE (2006)	Hybrid	Dynamic	Loose	N/A	Embedded	Unfilled	Unverified	One-to-one
High-Perf (2006)	Hybrid	Dynamic	Loose	N/A	Embedded	Unfilled	Unverified	One-to-one
SCUBA (2006)	Hybrid	Dynamic	Strict	CB-PR	Embedded	Unfilled	Unverified	One-to-one
Proactive (2007)	Software	Static	Loose	Sequential	Embedded	Filled	Unverified	One-to-one
Distributed (2007)	Software	Static	Loose	BB-PR	Embedded	Filled	Unverified	Many-to-one
SAKE (2008)	Hybrid	Dynamic	Strict	CB-PR	Embedded	Unfilled	Unverified	One-to-one
ReDas (2009)	Hardware	Dynamic	Loose	Sequential	Embedded	Unfilled	Verified	One-to-one
MobHat (2009)	Hybrid	Static	Loose	Sequential	Embedded	Unfilled	Unverified	One-to-one
DataGuard (2010)	Software	Dyanmic	Loose	Sequential	Embedded	Unfilled	Verified	One-to-one
SMARTIES (2010)	Hybrid	Dynamic	Loose	Sequential	Embedded	Filled	Erased	One-to-one
FPGA-Based (2011)	Hybrid	Static	Loose	Sequential	Embedded	Unfilled	Unverified	One-to-one
SMART (2012)	Hybrid	Dynamic	Loose	Sequential	Embedded	Unfilled	Verified	One-to-one
TrustLite (2014)	Hybrid	Dynamic	Loose	Sequential	Embedded	Unfilled	Verified	One-to-one
TRAP (2015)	Hardware	Static	Loose	Sequential	Embedded	Unfilled	Unverified	One-to-one
SEDA (2015)	Hybrid	Static	Loose	Sequential	Embedded	Unfilled	Unverified	One-to-many
C-FLAT (2016)	Hardware	Dynamic	Loose	Sequential	Embedded	Unfilled	Verified	One-to-one
DARPA (2016)	Hybrid	Static	Loose	Sequential	Embedded	Unfilled	Erased	One-to-many

2.4.1. Software-Based Software Attestation

This category describes attestation done entirely in software, making it suitable for legacy devices and requiring low cost to implement. Originally proposed in [49], and popularized in the SWATT algorithm presented in [8], where the device memory contains an algorithm which runs to check the remainder of the device memory. They also present generic requirements of any software-based software attestation algorithm, which includes pseudo-random memory traversal, resistance to replay attacks, high probability of detecting changes, small code size, optimized implementation, and non-parallelizability. These requirements are used to design the algorithm presented in Figure 14, which requires 23 machine cycles to execute each loop.

algorithm Verify(m)

//Input: m number of iterations of the verification procedure

//Output: Checksum of memory

Let C be the checksum vector and j be the current index into the checksum vector

for $i \leftarrow 1$ **to** m **do**

//Construct address for memory read

$A_i \leftarrow (RC4_i \ll 8) + C_{((j-1) \mod 8)}$

//Update checksum byte

$C_j \leftarrow C_j + (Mem[A_i] \oplus C_{((j-2) \mod 8)} + RC4_{i-1})$

$C_j \leftarrow rotate\ left\ one\ bit(C_j)$

//Update checksum index

$j \leftarrow (j+1) \mod 8$

return C

Figure 14: SWATT Algorithm [8]

An alternative to this algorithm uses self-verifying code. These algorithms store a verification function at a set memory location and run a checksum on the verification function itself [44]. Any alteration to the location of the function creates additional delay or an incorrect response. Figure 15 shows an implementation of this type of attestation algorithm, described as Indisputable Code Execution (ICE) [44].

//Input: y number of iterations of the verification procedure
//Output: Checksum C
//Variables: $[code\_start, code\_end]$ - bounds of memory address under verification
// $daddr$ - address of current memory access
// b - content of $daddr$
// x - value of T function
// l - counter of iterations
$daddr \leftarrow code\_start$
for $l = y$ to 0 **do**
  //T function updates x
  $x \leftarrow x + (x^2 \vee 5)$
  //Read from memory address a
  $b \leftarrow mem[daddr{+}{+}]$
  if $daddr > code\_end$ **then**
    $daddr \leftarrow code\_start$
  end if
  //Calculate checksum. Let C be the checksum vector and j be the current index.
  $C_j \leftarrow C_j + PC \oplus (b \oplus PC + l \oplus C_{j-2}) \oplus (x \oplus daddr + C_{j-1}) + PC$
  $C_j \leftarrow$ rotate left(C_j)
  //update checksum index
  $j \leftarrow (j + 1) \mod 8$
end for

Figure 15: ICE Algorithm [44]

2.4.2. Hardware-Based Software Attestation

This type of attestation aims to use tamper-resistant hardware to assist in software attestation. One category utilizes secure cryptoprocessors to execute cryptographic operations, providing tamper resistance to software. The secure processors include key generators, hash generators, random number generators, and encryption engines. A common example is the Trusted Platform Module (TPM). While the TPM primarily allows for systems to utilize a root of trust to ensure the system boots with the correct hardware and software configuration, one example of this category uses the TPM to add trusted time-stamping features to algorithms used in other software based approaches [10], [50]. Additional hardware solutions have been proposed to create a secure execution environment using executable only memory or instruction and memory encryption [46], [51].

2.4.3. Hybrid and FPGA-Based Software Attestation

This category relies on a combination of both general hardware and software to perform software attestation. In this context, 'hardware' also refers to reconfigurable hardware and FPGAs. Several examples utilize ROM to store code which allows for trusted software execution [52], [53].

One solution utilized an FPGA to read processor memory directly over a PCI bus to compute static attestations of running code on an Intel processor [54]. In a follow-on to that project, the authors implemented and tested a static attestation solution with a soft-core processor on an FPGA, an architecture which limits the functionality and portability of application code [55]. Despite the lack of a hard processor in the final results, their research presents a successful implementation of FPGA-based attestation and a secure communication channel.

Another application of FPGAs to software integrity research relies on inspection of processor memory accesses. These architectures aim to accomplish the goal of dynamic software integrity verification without directly scanning processor memory. Figure 16 presents one example, proposed by Zambreno et. al. [11]. Additional proposed architectures follow a similar framework [46], [47], [54], [56].

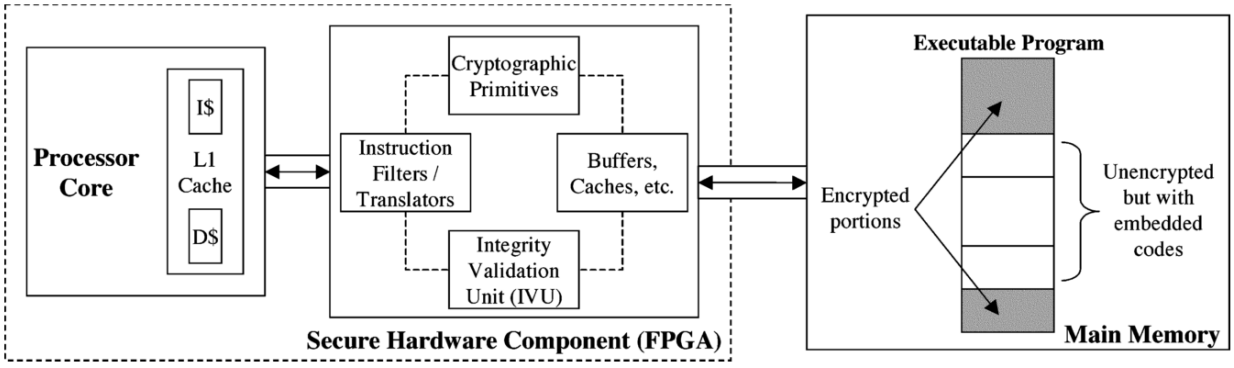

Figure 16: Software Protection Architecture [11]

By allowing the FPGA to monitor all memory accesses, these architectures present a means to analyze the processor for malicious actions. In addition, some frameworks propose encryption of data at rest [11]. Research utilizing these architectures commonly ends at simulation, due to the requirement for custom hardware solutions for implementation.

2.5. Limitations of Previous Research

Prior research in each method of software attestation suffered from multiple drawbacks. Successful attacks have been demonstrated against software-based software attestation [41]. Hardware-based software attestation requires complicated systems which may not exist in desired applications due to size, weight, and power (SWAP), or cost constraints. Hybrid attestation, while solving problems of timing and adding the capability of a remote trusted entity, still suffered from difficulty of implementation. For example, solutions which require custom compilers [47] present a high cost to design an attestation system. Alternatively, systems which utilize only FPGAs [55] require soft-core processors which significantly hinders the potential application speed. For example, the ARM-A9 completes nearly three times the MIPS/MHZ on the Dhrystone benchmark as the NIOS II soft processor. The solution presented in the following chapters utilizes the Cyclone V SoC architecture to create a solution which allows for simplicity, full hard processor functionality, and attestation speed not previously possible.

III. Design and Implementation

This chapter describes the design decisions made to utilize an SoC architecture to incorporate security into an embedded system through software attestation. First, a description of one use case is given, followed by a threat model to serve as motivation for the solution. A description of the proposed solution follows, with relevant assumptions and requirements. Alternative solutions are presented, and the potential architectures are categorized in an extended attestation taxonomy.

3.1. Motivating Example

This section presents an example software application that this research uses to demonstrate the benefit of using SoCs to incorporate security into embedded systems. Any additions must not alter the functionality of the example application, and all requirements must continue to be met.

Consider automobiles, which contain a large number of embedded systems to control varying components including the engine, the infotainment systems, and the brakes, to name a few. Each of these embedded systems are referred to as an Electronic Control Unit (ECU). ECUs complete various tasks throughout the car, based on their inputs and the messages they receive.

Similar to the functionality required by ECUs, the use case presented contains a microcontroller which reads inputs and gives commands to an external hardware device. The controller also contains a configuration and monitoring interface for a user to program functionality and monitor I/O. In the case of an automobile, another ECU may act as the user. On-board switches and LEDs simulate the inputs and outputs to/from a hardware device

respectively. The Cyclone V SoC on the Arrow SoCKit board simulates the controller for this use case, with ARM microprocessors increasingly used in automotive applications [57]. Figure 17 shows the SoCKit development board used in this research.

Figure 17: Arrow SoCKit Development Board [58]

In this example application, the Cyclone V FPGA remains unused, to simulate a design which relies solely on a microcontroller. The use case design is based on the Golden System Reference Design (GSRD) [59], often used for beginning new designs on the Cyclone V. The design configures various components on the SoC through the configuration mode shown in Figure 7 (HPS boot first), described in the following steps:

1. HPS reads the boot select pins, in this case set to boot from SD card.
2. HPS transfers the preloader from the SD card to SDRAM and runs it.
3. The HPS transfers the bootloader (u-boot) from the SD card to SDRAM and runs the bootloader, in this case containing a script to read additional files and load the kernel.

4. The HPS boots into Angstrom Linux and starts services which include HTTP, SSH, and others.

This default configuration is used for this example. After booting Linux, the user runs an executable located on the HPS compiled from C code to read switch positions, complete various calculations, and output switch values to LEDs (Section 3.1.1 addresses this application in detail). In this use case, the user configures the functionality of the application through an executable loaded onto the board through SCP and monitors the application through UART or SSH. In an automobile, these interfaces would likely be implemented by a control area network which allows the same functionality.

Figure 18 shows the component legend which carries through the remainder of this research to indicate various user and attestation signals, as well as exploited components and encryption.

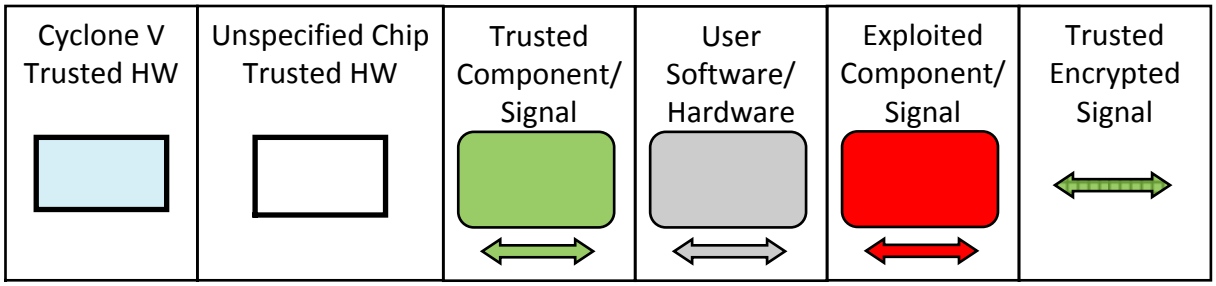

Figure 18: Legend for All Following Block Diagrams

Once configured, the system operates as shown in Figure 19, with a "controller" configured to operate on inputs (switches) and produce outputs (LEDs), and a "user" who acts as a remote monitor to send commands, make changes, and observe status as necessary. The term "interface" represents any protocol or application used to interact with the controller. In this example, the interfaces GCC and SSH serve to compile an application and load and run it, respectively. The user configuring the controller relies on trusted hardware in the

microprocessor and the memory controller to be free of malicious functionality at the hardware level (see Section 3.3.2). Figure 19 shows the network described above, where a generic microcontroller runs an application stored in off-chip memory to communicate with IO and a configuration interface.

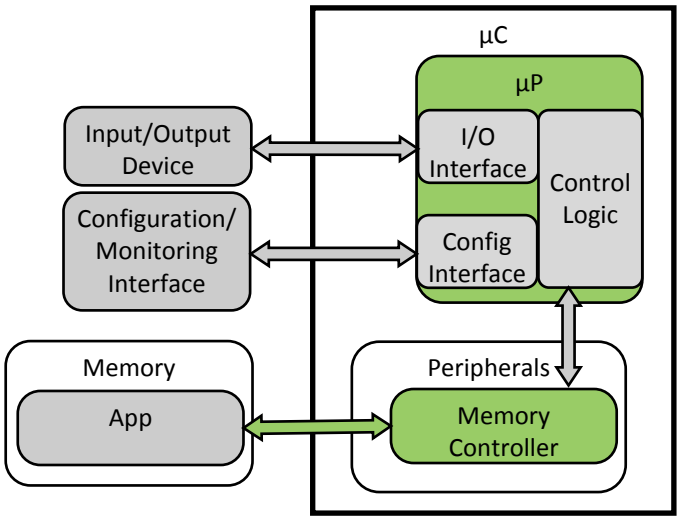

Figure 19: Use Case Block Diagram Utilizing Generic Microcontroller

In this case, the user also relies on the potentially compromised application code and the output from the application, without having any way to verify them. For example, the Stuxnet computer worm overwrote organization blocks of PLCs while operators, unable to physically observe the change in functionality, trusted that PLCs operated in accordance with the feedback from the configuration interface [4]. This single point of failure presents the weakness this research intends to address.

3.1.1. Application Functionality

The motivating example for this research focuses primarily on floating point calculations. As a result, the Whetstone benchmark fits this requirement better than other common benchmarks [60]. In comparison, the CoreMark benchmark focuses on cache operations, and the

Dhrystone benchmark primarily focuses on integer and branch operations [61]. This benchmark runs as part of an example application which carries through the remainder of this thesis. The application completes two stages:

1. Initial Hardware Configuration

This stage sets the hardware according to the desired properties of the application or the test. The capabilities of this configuration range from L1 and L2 cache enable and disable, bus priority assignment, timer configuration and start, and GPIO and clock assignments, variable instantiation and initialization, and anything that the application only requires once.

2. Loop

This stage loops until the user stops the application. Within each loop, the microcontroller completes the following functionality, commonly required of embedded systems:

a. Read Input. This amounts to reading from hardware registers that map to switch values and storing this value.
b. Complete Computations. A reduced number of loops on the Whetstone benchmark runs at this stage.
c. Set Output. This stage sets the value of the on-board LEDs to the switch value read above by writing to hardware registers that map to LED outputs.

Figure 20 shows the output of the monitoring application, with simple print statements to notify of status. Figure 21 shows an example of the visual feedback of the inputs and outputs of the application. Note that the right side of Figure 21 shows the FPGA switches and LEDs, unused at this time. This required functionality caries through the remainder of this thesis.

```
-------------------------
Reading switches...
Completing computations...
Setting LEDs...
-------------------------
Reading switches...
Completing computations...
Setting LEDs...
-------------------------
Reading switches...
Completing computations...
Setting LEDs...
-------------------------
Reading switches...
Completing computations...
Setting LEDs...
-------------------------
Reading switches...
Completing computations...
Setting LEDs...
-------------------------
```

Figure 20: Application Output

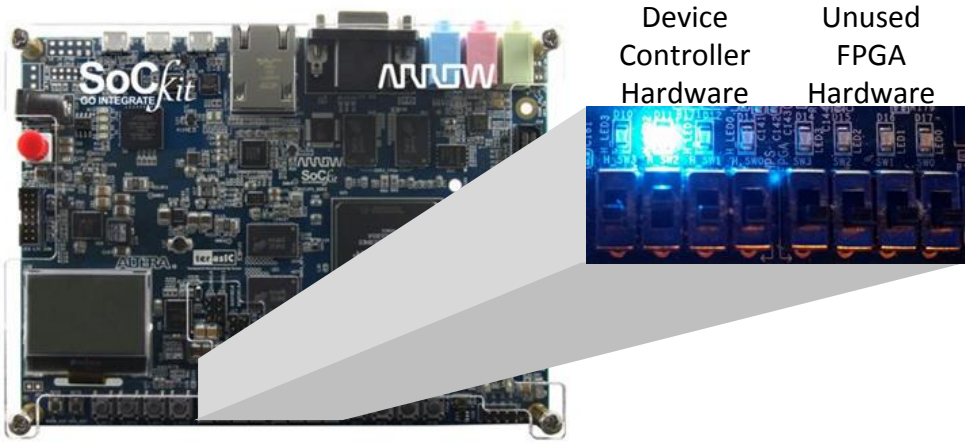

Figure 21: Example of Device Controller Functionality (Left Switches and LEDs) and Unused FPGA Resources (Right Switches and LEDs)

3.1.2. Application Requirements

3.1.2.1. Real-Time

The application developed requires response time to remain within an acceptable window. For purposes of this research, specific time requirements are not addressed. However, a realistic security implementation must minimize impact on application response time. The amount of acceptable delay depends on the application.

3.1.2.2. Reconfigurable

It is assumed that the application developed requires reconfigurability. While one-time-programmable solutions solve many security challenges [13], a decrease in flexibility prohibits their use in many situations. A realistic and more general security implementation will permit the user to alter functionality.

3.2. Threat Model/Potential for exploitation

An attack model motivates the need for increased security. An attacker, their capabilities, and their knowledge are described, followed by an example of potential exploitation steps.

3.2.1. Attacker Goal/ Knowledge

In the scope of this thesis, an attacker has the goal of altering the execution of the embedded system, while remaining undetected to users who configure/monitor the system. The threat model is broken up into two components: technical and operational capabilities, as in [62]. Technical capabilities represent what the adversary knows about the destination system. This category may include the following:

- the operating system and application versions running,
- all binaries of firmware and source code running on device controllers,
- configuration modes and location of static and dynamic data, and
- any additional design choices.

This research assumes the attacker has access to a fully operational system and all available information about that system. In the context of advanced threats, it is realistic to assume the adversary has a working model of the system, with all accompanying documents [4].

Operational capabilities represent what the adversary has the ability to access on the final system. This category may include the following:

- physical access to either device controllers or physical network infrastructure,
- local proximity to system, access to wireless network, and
- network access through an air gap, gained through USBs, supply-chain sources, etc.

This research assumes the attacker gains access to the network used to configure the microcontroller, but lacks physical access to the embedded system. On automobiles for example, infotainment systems may be accessed, but sensitive engine ECUs remain secure. Given an attacker with these capabilities and knowledge, the following section presents an attack methodology the attacker may follow to compromise this system.

3.2.2. Attack Steps

The attacker gains access to the network used to program the microcontroller. This access may be gained through diagnostic ports, infotainment systems, or any number of zero-day exploits. For example, the Stuxnet worm may have been introduced to nuclear facilities through compromised removable drives [4]. Figure 22 shows the status of the attacker at this point, with an attacker who compromises the configuration network and who can now directly interact with any microcontroller interfaces in use.

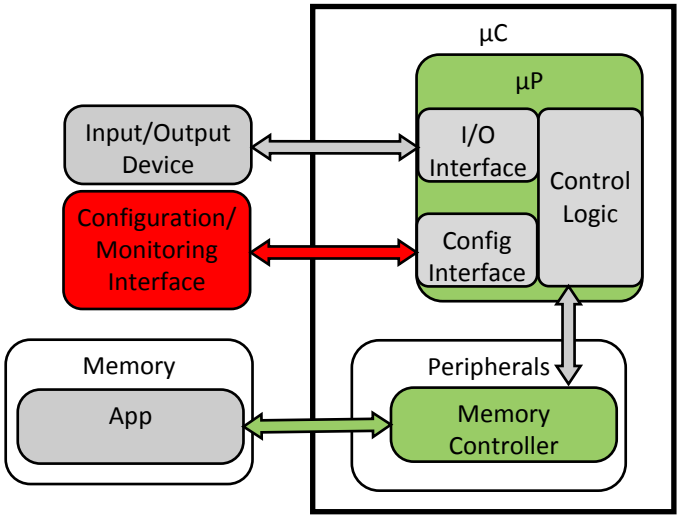

Figure 22: Attacker State

With the configuration network compromised, the attacker exploits or gains access to the interface used to configure the controller. For example, credentials obtained on the network may allow SSH access, or an HTTP server or listening application may contain security vulnerabilities. This simulates a malicious software attack, a common attack against embedded systems [63]. After compromising the interface to gain access, the attacker alters the software running on the microcontroller. The system continues to appear to operate as normal, failing to alert the user to the change in functionality. A similar procedure has been used by advanced malware threats [4].

At this point, system integrity is lost. The user has no way to further verify running behavior, except looking at the (now compromised) monitoring interface or the physical output, which may not be feasible. Ultimately, the user inappropriately places trust in an interface which they unknowingly have no remaining control over. Figure 23 shows the final state of the compromised network.

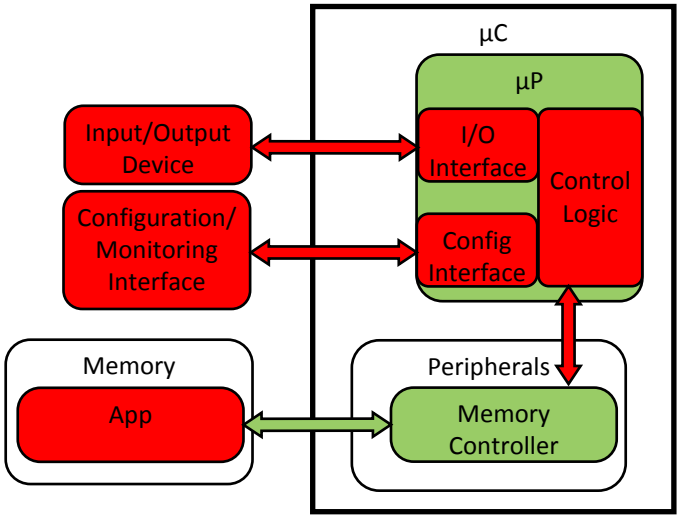

Figure 23: Final Attacker State

Knowing that the potential for this form of attack exists [4], the next section begins the discussion of a solution to this threat by presenting assumptions inside which a solution would present a similar attack from succeeding.

3.3. Assumptions

3.3.1. Physical and Communication Security

The solution presented may fail to secure systems which lack physical security. In special publication 800-82, the National Institute of Standards and Technology recommends physical security for industrial control systems, stating that:

> Gaining physical access to a control room or control system components often implies gaining logical access to the process control system as well. Likewise, having logical access to systems such as main servers and control room computers allows an adversary to exercise control over the physical process. [64]

Simply put, as Microsoft writes in their ten immutable laws of security: “If a bad guy has unrestricted physical access to your computer, it's not your computer anymore” [65]. The solution presented assumes an adversary has no physical access to the system.

In addition, this assumption includes security of communication between the external verifier and the FPGA. Note this communication is separate from the communication between the device, the controller, and the user, which are assumed insecure.

3.3.2. Hardware Trust

Vital to the security of any hardware system is trust in the hardware. In the context of this thesis, this means a lack of any unintended features, as well as the accuracy of any manufacturer technical documents. Malicious hardware trojans may be inserted at any stage from initial development to final assembly, and can potentially trigger on a variety of conditions [66]. For example, in 2012, it was discovered that a backdoor existed in military chips that granted those with knowledge of its functionality the ability to compromise secret keys and configuration data [67]. To combat this threat, The NSA created the program on Trusted Foundries, and DARPA created the Trust in Integrated Circuits Initiative [68]. The threat of undesired hidden functionality in hardware is assumed to not exist here, although realistic systems should follow appropriate guidance according to their security posture to minimize this threat.

3.3.3. Trusted Verifier

Due to architectural limitations addressed in Section 2.1.2.1, any solution relying on the Cyclone V FPGA to incorporate security into the Cyclone V HPS requires an external trusted

verifier. The specific implementation of the trusted verifier is not set here. As an example, a few options are presented in Section 3.5.3. For any implementation chosen by the user, it is assumed that while the configuration network may be insecure, the verifier is trusted.

3.3.4. Trusted Programmer

In order to incorporate trust in an SoC device through the addition of an FPGA, a similar problem to a microcontroller only system presents itself: a PC is required to program the FPGA. An adversary with access to the computer used to program the FPGA who replaces code or otherwise inserts functionality into the hardware breaks the system integrity. Extra care must be taken to secure the computer used to program any FPGA components. This computer must remain distinct from the configuration network and has no need to communicate on any network. The solution discussed assumes all FPGA programming files and functionality are trusted.

3.3.5. Golden Copy

The trusted verifier requires knowledge of expected responses to determine the validity of device responses. In order to accomplish this, the trusted verifier must possess a copy of the expected program, assumed to free of malicious functionality.

3.4. Design Decisions and Implications

This section presents the decisions made when choosing how to incorporate security while replacing a microcontroller only system with an SoC. A description of the proposed architecture is presented. Then, for each decision area, options are presented, followed by the choice made as well as the security implications of making that choice.

3.4.1. Overview

The decision to utilize an SoC grants multiple benefits over a microcontroller only system. For the additional cost of hardware and design effort, SoCs present the opportunity to incorporate security in ways that are otherwise not possible with microcontrollers:

- Flexibility in boot choices allows for integrity checking at every boot stage.
- Any programs stored in processor RAM are directly accessible to the FPGA.
- A variety of communication bridge options allows for flexibility in architectural design decisions.
- The FPGA security implementation remains entirely external to processor functionality, allowing for attestation at runtime.

These benefits are leveraged in a solution that relies on the Cyclone V FPGA as a root of trust to monitor the microcontroller functionality and alert to any deviation from expectation. Figure 24 shows the architecture proposed, with trust now stored in the FPGA. Each aspect of the architecture, attestation, and boot is discussed in the following sections. The primary contribution of this architecture is the utilization of SoC hardware to incorporate attestation logic to dynamically inspect the processor memory for unintended alterations.

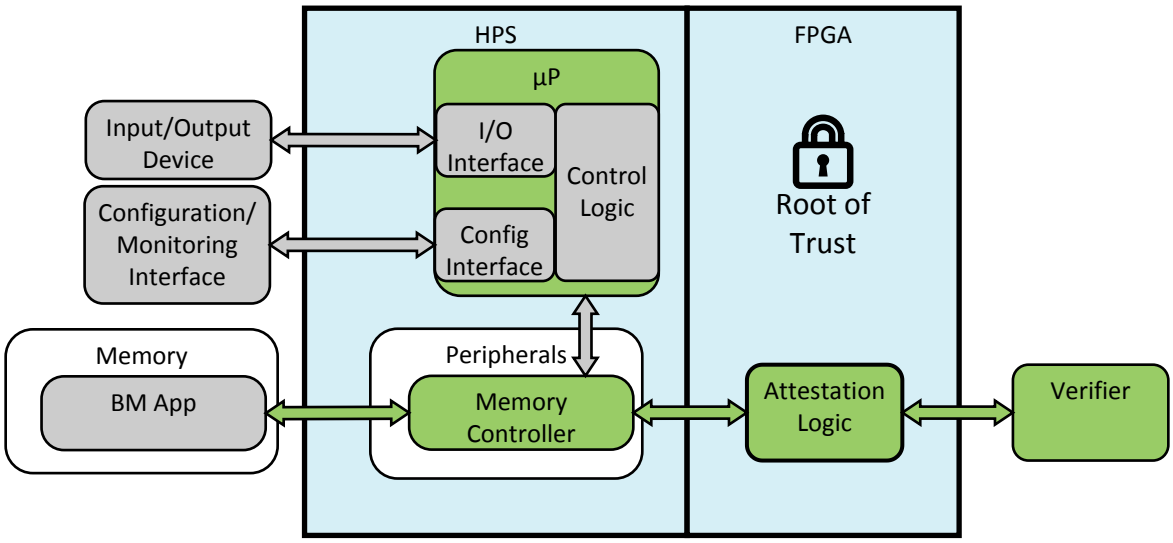

Figure 24: Proposed Solution Architecture

Error! Reference source not found. shows the procedure used to power-on the device. he trusted operations are shown in the dashed area. The FPGA configures with a bitstream created by the user, which contains a preloader downloaded by the processor. Once fully configured, the FPGA begins to run attestation on the application the processor downloaded from external non-volatile memory (NVM).

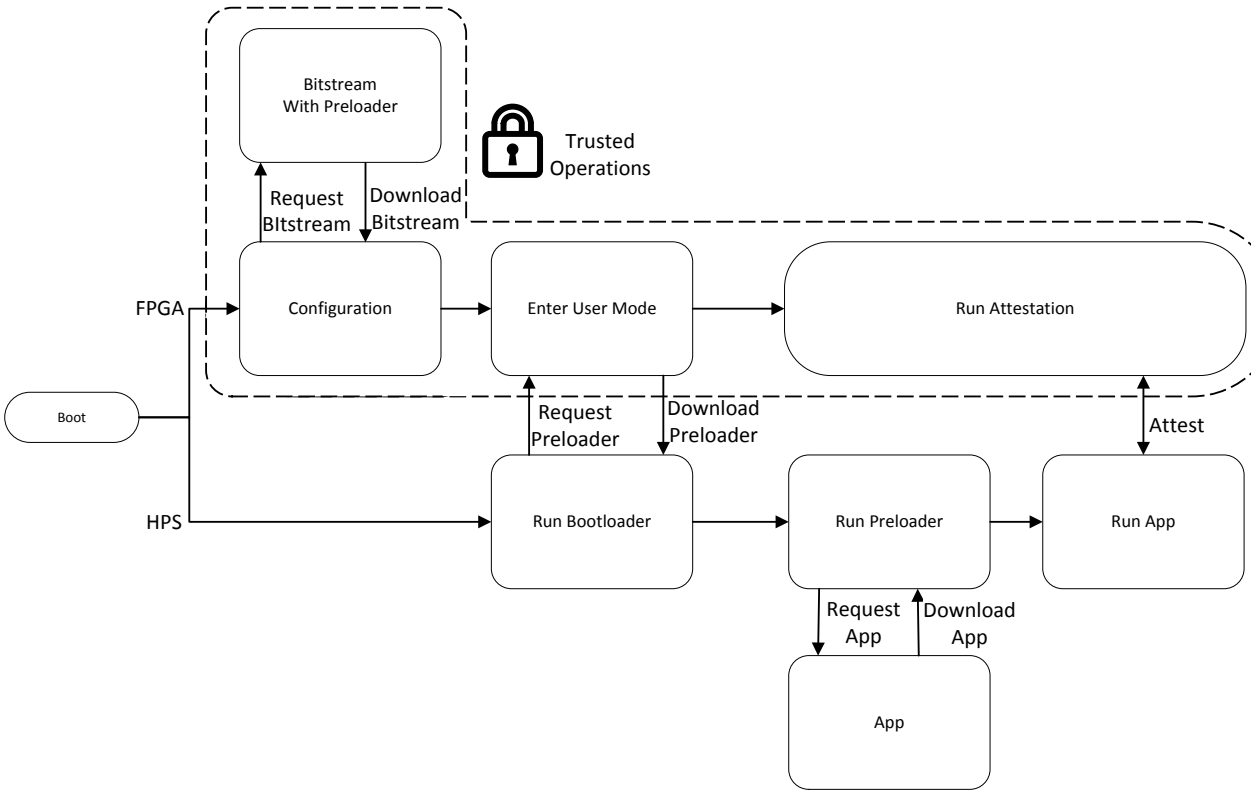

Figure 25: Proposed Solution Boot Procedure

Figure 26 shows the generalized state machine for attestation logic in the SoC. The option exists to incorporate various additions including bursting, encryption, and signatures. The FPGA receives a request from the external verifier, potentially decrypts it, computes the checksum according to the chosen algorithm, and responds to the verifier with a potentially encrypted checksum.

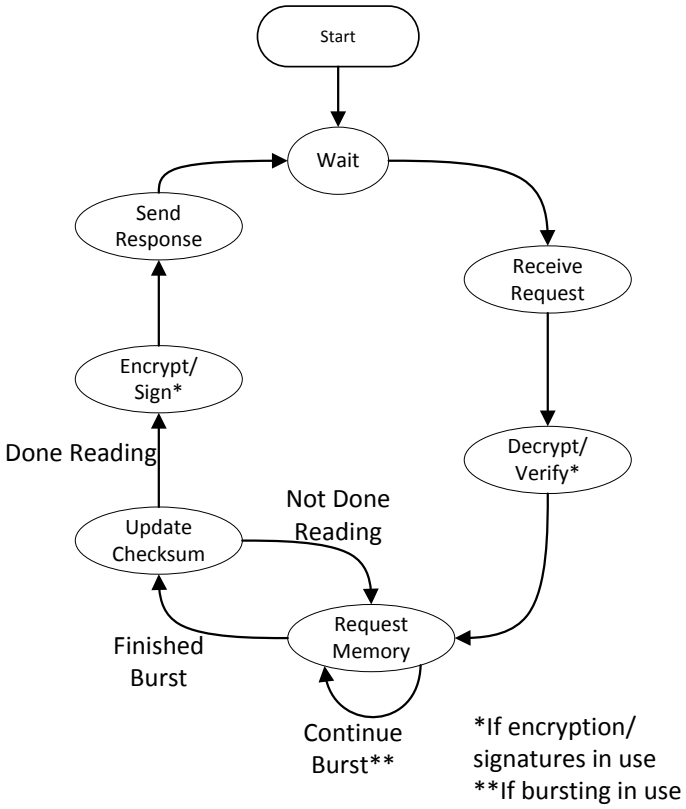

Figure 26: Proposed Attestation Logic Finite State Machine

3.4.2. Boot Procedure

The boot procedure of the SoC should be chosen to give the FPGA as much control as possible while continuing to meet functional requirements. In the Cyclone V SoC, there are three means of boot, described in Section 2.1.2.3. The decision between these options presents significant security tradeoffs for the final application [69]. Table 2 compares these options in the context of a desire to increase system security through the addition of the FPGA. NVM options include SD cards, NAND, or QSPI sources, and belong exclusively to either the HPS or the FPGA side of the Cyclone V.

Table 2: Boot Options

Boot Summary	HPS and FPGA boot separately	HPS boots first, configures FPGA	FPGA boots first, configures/programs HPS
HPS Preloader Source	HPS-only NVM	HPS-only NVM	FPGA-only NVM
HPS Bootloader/Bare Metal Source	HPS-only NVM	HPS-only NVM	FPGA-only NVM or HPS-only NVM
FPGA Configuration Source	FPGA-only NVM	HPS-only NVM	FPGA-only NVM
Pro	1. Flexibility 2. Loose coupling	1. Dynamic FPGA functionality	1. Known HPS secondary boot location 2. HPS cannot reprogram FPGA 3. Potential boot-time attestation
Con	1. Unknown secondary boot location	1. HPS can reprogram FPGA	1. Flexibility

This framework uses the last mode, where the FPGA boots first, and then configures the HPS. This grants the following benefits:

- The FPGA stores the preloader used to configure the HPS, guaranteeing trusted knowledge of HPS secondary program location and enabled bridges and interrupts. These benefits are expanded upon further in the following sections.
- The FPGA bitstream is inaccessible to the HPS, preventing a malicious program from attempting to access or alter FPGA functionality.
- FPGA reconfiguration is impossible from the HPS, preventing a malicious program from fully reprogramming the FPGA. Any attempt to do so from the HPS simply forces FPGA to reconfigure with the same external bitstream. The implications of this are addressed further in Section 3.5.3.

- The potential to store both the HPS preloader and program in the FPGA bitstream allows for the possibility of boot-time attestation. This topic is addressed in Section 3.4.7.

3.4.3. Preloader Storage

As mentioned above, the proposed architecture stores the HPS preloader in the FPGA. The preloader is responsible for completing the following sequence of actions:

1. perform HPS initialization, including bridges and clocks,
2. initialize SDRAM interface,
3. load next boot stage into SDRAM, and
4. jump to the next boot stage.

Storing this file in the trusted FPGA bitstream guarantees that the system boots into a state that allows for the FPGA to have access to HPS memory, know program storage details, and have the ability to reprogram the HPS. If an untrusted source hosts the preloader for the HPS, the location of the next stage of execution is unknown and vulnerable to change by a malicious program. Knowing the next stage of execution grants the FPGA the ability to monitor that location for content, enabling an attestation capability further discussed in Section 3.4.6.

3.4.4. FPGA Programming

As mentioned in Section 3.4.2, booting the FPGA first removes the ability for the HPS to access FPGA configuration data before configuration. Additionally, regardless of boot procedure, the HPS never has access to read back configuration data from the FPGA. The only means the HPS has to access FPGA configuration data is if the HPS stores the configuration file

used. This knowledge, along with the assumption of physical security, removes any need for implementing encryption in the FPGA bitstream, eliminating key-management challenges. Figure 27 shows the architecture of the FPGA configuration in the Cyclone V, and the FPGA control block interface that only supports data inputs from the HPS [17].

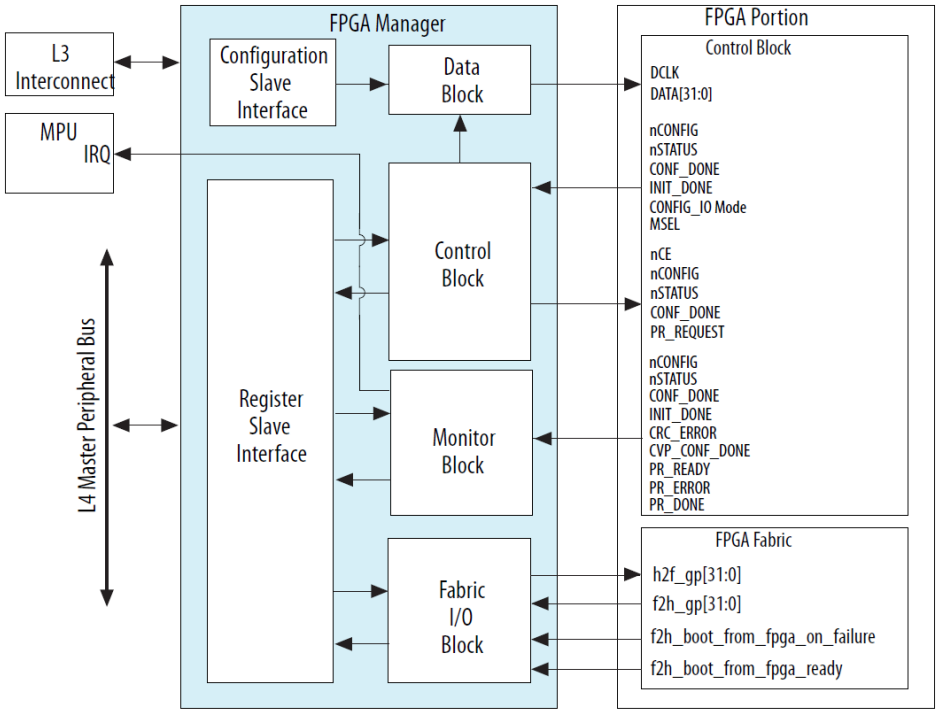

Figure 27: FPGA Configuration Architecture [17]

3.4.5. Communication Bridge

Section 2.1.2 introduced several bridges available in the Cyclone V SoC. From the perspective of incorporating security with the SoC, Table 3 presents each bus with pros and cons to relying on their use.

Table 3: Comparison of SoC Communication Bridges

Bridge	HPS-FPGA	LWHPS-FPGA	FPGA-HPS	FPGA2SDRAM
Master	HPS	HPS	FPGA	FPGA
Protocol	AXI/Avalon	AXI/Avalon	AXI/Avalon	AXI/Avalon
Width	32,64,128	32	32,64,128	32,64,128,256
Pro	1. Simplified APIs	1. Simplified APIs 2. Faster than H-F	1. FPGA acts as master	1. Direct Memory Access 2. Uninterruptable 3. No HPS software required 4. FPGA acts as master 5. Configurable priority
Con	1. User logic req 2. HPS acts as master	1. User logic req 2. HPS acts as master	1. User logic req	1. Memory controller bottleneck 2. No access to some regions

Given these choices for communication between the SoC components, the architecture given in Figure 24 utilizes the FPGA2SDRAM bridge with a 256-bit bus. This allows for the FPGA to master memory accesses directly to the memory controller. The HPS has no ability to monitor or interrupt these accesses. This fact, paired with external verifier communication which does not pass through the microcontroller means that common attacks against software attestation including rootkits and compression have no ability to trigger when attestation starts [41]. A potential disadvantage to this choice is the bottleneck of utilizing a single memory controller component to feed the processor requests, as well as the FPGA requests. The architecture which causes this potential problem is shown in Figure 28, where the memory controller acts as an arbiter for the single-port backend.

On a real-time system, delays in response time may cause the system to fail to meet deadlines. The impact of this choice on system response time is addressed in experiments proposed in Chapter IV. Another drawback is the lack of FPGA access to certain regions of processor memory. This limitation is addressed in Section 6.7.2.

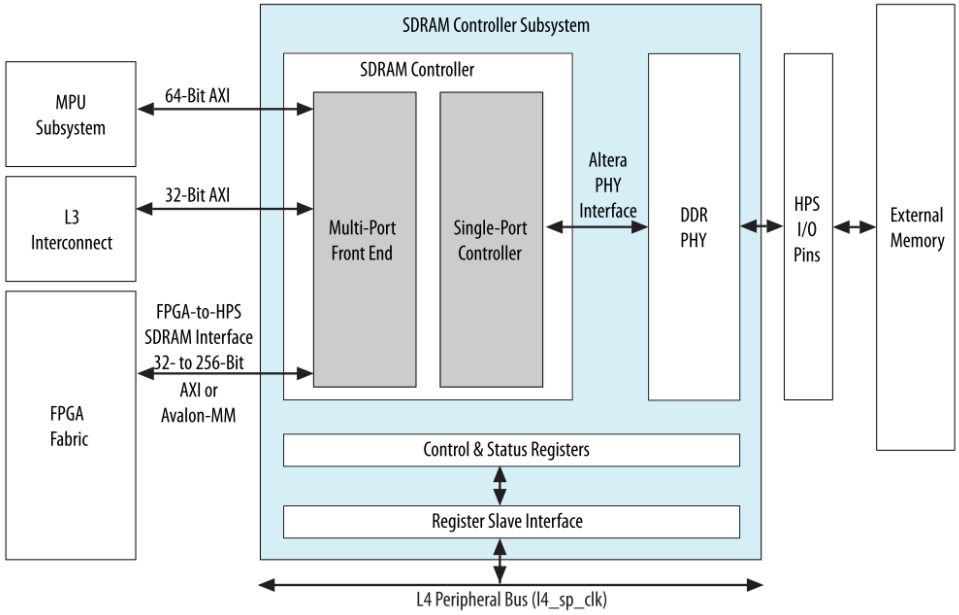

Figure 28: SDRAM Controller Block Diagram [17]

3.4.6. Bare metal app and attestation enabling

When compared with a Linux operating system like that shown in Section 3.1, bare metal applications complete tasks faster, and with less jitter [70]. In addition, bare metal applications are compiled for the destination system, loaded in a predetermined address, are the only application running, and directly reference static memory. This allows for the external attestation of code run on the processor from known locations in memory [8]. Section 3.5 covers the various aspects of this attestation in detail. In addition, bare metal applications can be more completely analyzed, removing any unnecessary functionality to further prevent exploits such as return-to-libc attacks or Return-Oriented Programming (ROP) chains [71]. As a result of both of these benefits, transitioning a system from a Linux operating system to bare metal potentially improves performance and security. The proposed solution utilizes a bare metal application which accomplishes the same functionality presented in the use case. While not completely secure, how an attacker could utilize a bare metal application interface to dynamically insert

malicious functionality is beyond the scope of this research, although research proves the potential for exploitation [9]. A simulated attack is presented in Section 6.2.1.

3.4.7. Optional Program Storage and Boot-Time Attestation

Section 3.4.2 discussed the options for booting the Cyclone V SoC. When utilizing the FPGA to store a preloader for the HPS, the HPS second stage often resides in another location, such as an SD card. This is not a requirement, and the HPS may subsequently run a bare metal application that is also stored in the FPGA fabric [19]. In this case, as long as the application fits in the 10MB on-chip memory available or another device sends the application to the running FPGA, the FPGA possesses the ability to integrity check the potentially encrypted application before HPS programming at each power-on, enabling boot-time attestation without the use of secure boot technologies.

3.5. Attestation Logic

Software attestation, introduced in Section 2.2, increases security through integrity checking of code in embedded systems. In this example, the embedded system running a bare metal application meets the requirements for implementation of software attestation. The following sections discuss the implementation of FPGA-based software attestation in the system described in Section 3.1.

3.5.1. Attestation Algorithm

The attestation algorithm presented communicates through the memory controller bridge described in Section 2.1.2. Due to the fact that the HPS has no ability to monitor memory

requests from the FPGA, a simple attestation algorithm is used. This algorithm is described in the following figure. This algorithm takes as input an address range 'A', and an initial checksum vector. The attestation logic linearly scans the address range in increments of 256 bits, starting at the given initial value. The FPGA reads values from the HPS memory controller and performs ADD and XOR operations on memory as well as previous checksum values, making the checksum computation non-associative [44]. This denies pre-computation attacks, ensures that a single byte perturbs additional checksum values, and that order matters in the computation. Once completed, a resultant checksum value notifies the verifier of the status of the result. The security of the algorithm used here is analyzed in [8]. The only difference being that in this implementation, using trusted communication directly to the FPGA ensures the processor has no ability to redirect requests, nor any ability to predict or redirect any memory request. As a result, pseudo-random reading of memory is not required, and this algorithm safely utilizes a block-based sequential scheme.

Compute Attestation Using Add and XOR

INPUT: Address range A, Starting address A_0, Initial Checksums C_1, C_0
OUTPUT: Final Checksum C

1: **for** $A_i \in A$ **do** //Read Linearly, $i = 2, 3, 4..$
2: $V \leftarrow readAddress(A_i)$
3: $C_i \leftarrow C_{i-1} + (V \oplus C_{i-2})$
4: **return** C

Figure 29: Attestation Algorithm

While the FPGA computes the attestation checksum, the verifier either looks up the expected value from an array of precomputed results, or potentially calculates the expected result on a local trusted memory space. Figure 30 shows the full communication protocol. The assumption of physical security grants a simplified procedure here, as the communication

between the verifier and the FPGA requires no encryption. If desired, encryption and authentication can be added to this protocol, as in Section 6.4.2.

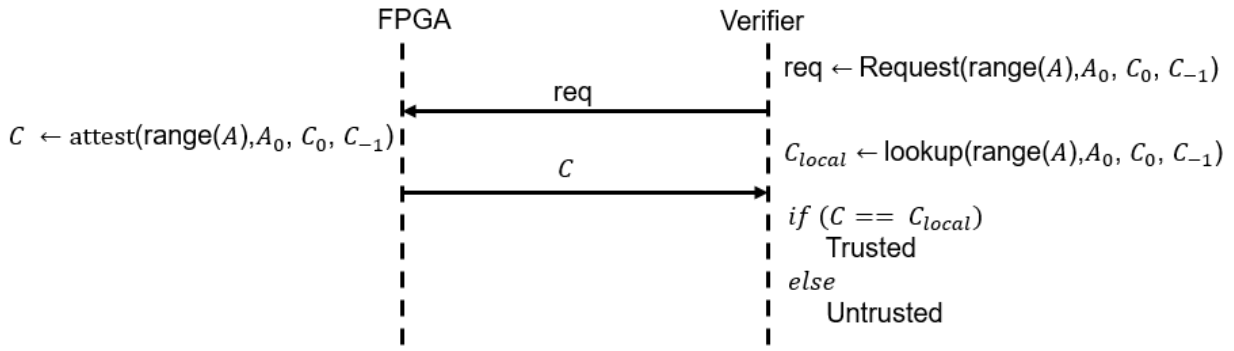

Figure 30: Attestation Communication Protocol

This algorithm represents a simple and fast choice for attestation, and allows for detection of a single byte of data alteration at any memory address. This research makes no claims about the cryptographic strength of this algorithm; cryptographic security analysis is out of the scope of this research. More complex attestation options are addressed in Section 3.5.3. Whichever attestation method is desired, this framework allows designers to construct their own algorithms to attest the memory space.

3.5.2. Memory Space to Attest

Before beginning attestation, the external verifier needs to define the address space to attest. It is important to note that the addressing space for the three elements is not necessarily the same. Figure 31 identifies the memory mapping for the processor to the main memory SDRAM memory. The key element in this figure is the MPU SDRAM Window, which may expand or contract, and maps directly to the FPGA-to-SDRAM address range. The spaces that do not translate to this SDRAM region either contain boot code, on-chip RAM, or map to FPGA slaves and peripheral control registers.

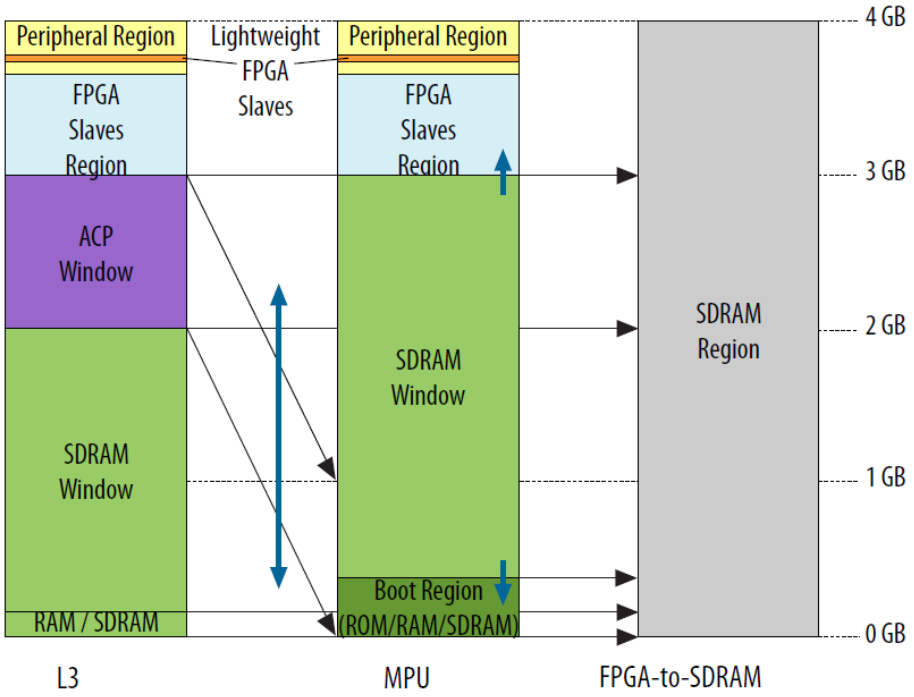

Figure 31: Cyclone V HPS Memory Map [17]

By default, the processor has access to the "SDRAM Window", from 256KB to 3GB. Through configuration, the processor may enable access to the remaining 1GB of SDRAM region space. Only the instruction memory of the processor bare-metal application is attested in this solution, with a size dependent on the application under test, known by the size of the golden copy.

Figure 32 shows the MPU address space, before and after boot. The peripheral region contains control registers for the processor, and the FPGA Slaves Region contains interfaces to slaves configured in the FPGA, unused here. The rest of the space maps to either unused application regions, or static or dynamic application memory regions. The red '.bin' region contains the executable instructions of the application, being attested in this architecture. The rest of the memory space is either: unreadable by the FPGA or should be marked Non-eXecutable (NX).

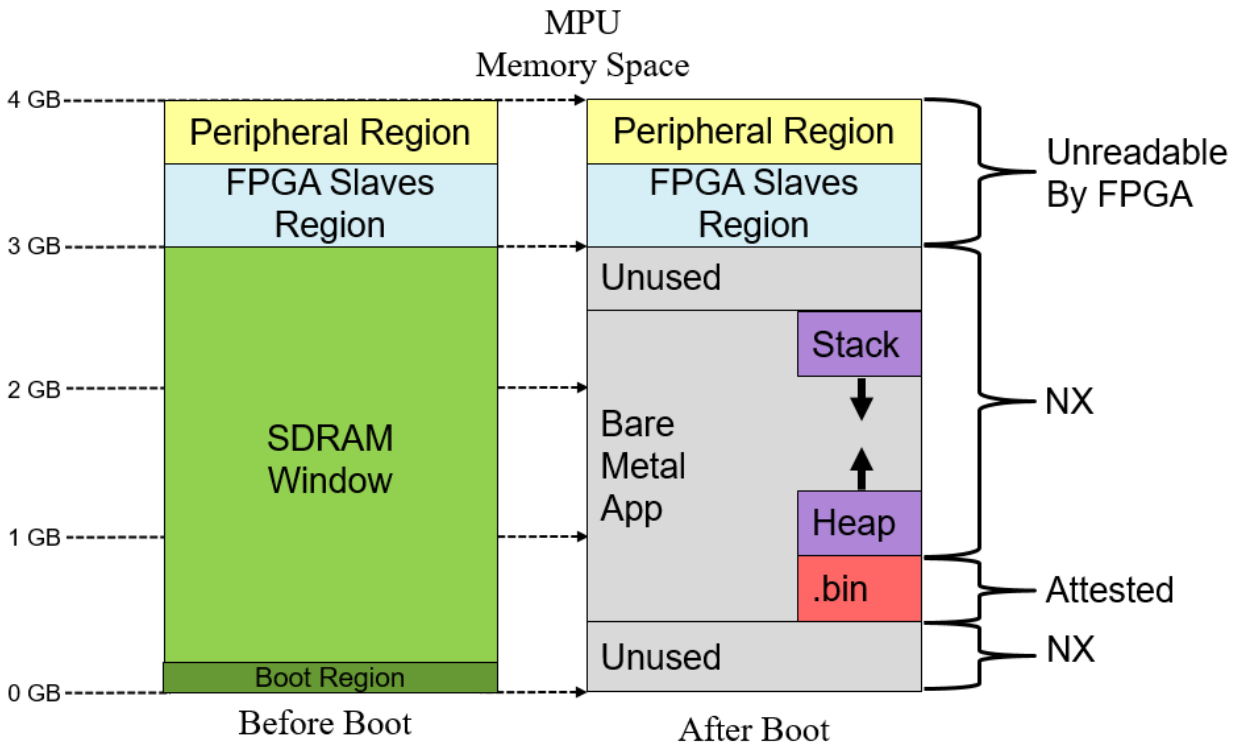

Figure 32: Attestation Address Space

One potential software exploit, a buffer overflow, could attempt to execute instructions placed in the stack or heap, areas not included in this attestation. To combat this threat, attestation requires executable space protection through setting an NX bit, or through filling additional memory with random data [72]. As a result, an adversary would require a more complicated attack to corrupt the instruction memory, which would still alert the attestation. The Cortex-A9 processor lacks an NX bit, a limitation detailed in Section 6.7.3.

The architecture presented allows designers to expand this attestation to additional memory spaces and algorithms. For example, an algorithm could be designed to analyze the entire memory space of the SDRAM window to include the dynamic regions (see Section 7.4.2.1). Additionally, the FPGA could utilize the SDRAM bridge or the preloader to write to unused processor address spaces in addition to attesting them. This would prevent an adversary

from attempting to conceal data or instructions in any unused portions of memory, an attack presented in [8].

3.5.3. Alternate Attestation Algorithms and External Verifier Options

This chapter presented a framework which includes a simple attestation algorithm which can alert to memory alterations. In this solution, the external verifier acts as a simple comparator, checking for the value of a checksum output from the FPGA and comparing it against stored expected values. This simple algorithm allows for the fastest attestation of the entire memory space, where memory can be read and incorporated into the attestation checksum in a single FPGA clock cycle. This allows for testing of the worst-case effects of this attestation on the HPS functionality at runtime as well as maximum speed for memory reading, proposed in Chapter IV.

More complex attestation algorithms include additional cryptographic primitives for security in the checksum and communication methods. For example, SWATT incorporates randomness to the order of memory spaces read [8]. Their algorithm incorporates the RC4 cypher into the attestation address calculation to read data from pseudorandom addresses. The algorithm, which may increase security to the attestation, also increases runtime. For example, RC4 takes several clock cycles to process a byte of data. Additional attestation algorithms require SHA calculations for the checksum and encryption for all communication, forcing additional delays in computation.

Note that random reading of memory is required in a system where the processor could potentially predict when memory spaces are read [8], and use collusion or memory copy attacks to beat the attestation [73]. In the framework presented, the processor has no knowledge of

attestation start or memory reads, negating any requirement for random memory reads. If random reading is desired is this system which utilizes bursting transactions, block-based pseudorandom memory traversal is required. Security of block-based pseudorandom reading is addressed in [73].

While these cryptographic algorithms (ciphers, hashes, encryption, etc.) increase the runtime of the attestation algorithms, the memory controller may still present the bottleneck in this system. If the memory controller consistently takes dozens of clock cycles to respond, hashing and encryption may present no significant decrease in runtime to the algorithm.

The following sections address potential additions and alterations to increase security of the checksum logic, as well as the communication channel between the verifier and the FPGA, and the potential utilization of the HPS.

3.5.3.1. For Increased Checksum Security

The security of the algorithm presented previously lacks formal proofs of security, despite its use in previous attestation solutions [8]. As a potential improvement to increase the cryptographic security of the checksum, commonly accepted hash algorithms could be incorporated (SHA2/3, MD6, etc.). The attestation logic would the compute a hash on the memory space of the HPS, and return the value to the verifier. The reliance on proven hashing functions reduces the potential for collisions. Figure 33 shows the architecture of this potential solution.

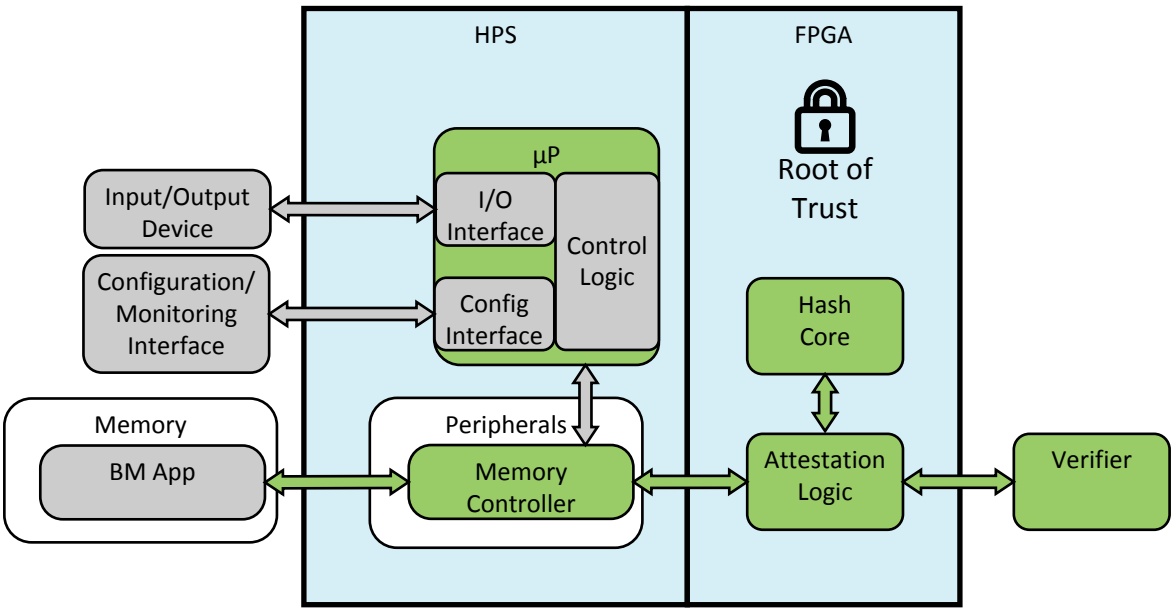

Figure 33: SHA256 Checksum Block Diagram

Yet another option removes randomness and cryptographic functionality, while increasing security in the attestation at the cost of flexibility and FPGA resources. The unique nature of FPGA design allows for fetch and compare of data values in a single clock cycle, enabling a direct memory comparison, shown in Figure 34. The application binary may be stored entirely in the FPGA bitstream, where the attestation logic can compare every byte in the stored code with the local golden copy. Any discrepancy would immediately create an alert and notify of location, as opposed to being forced to wait for the calculation of an entire checksum, only to lack knowledge of location of exploited code. The limitation for this framework is the requirement that the entire application binary fits into on-chip memory, which means less than 10MB on the Cyclone V SoC.

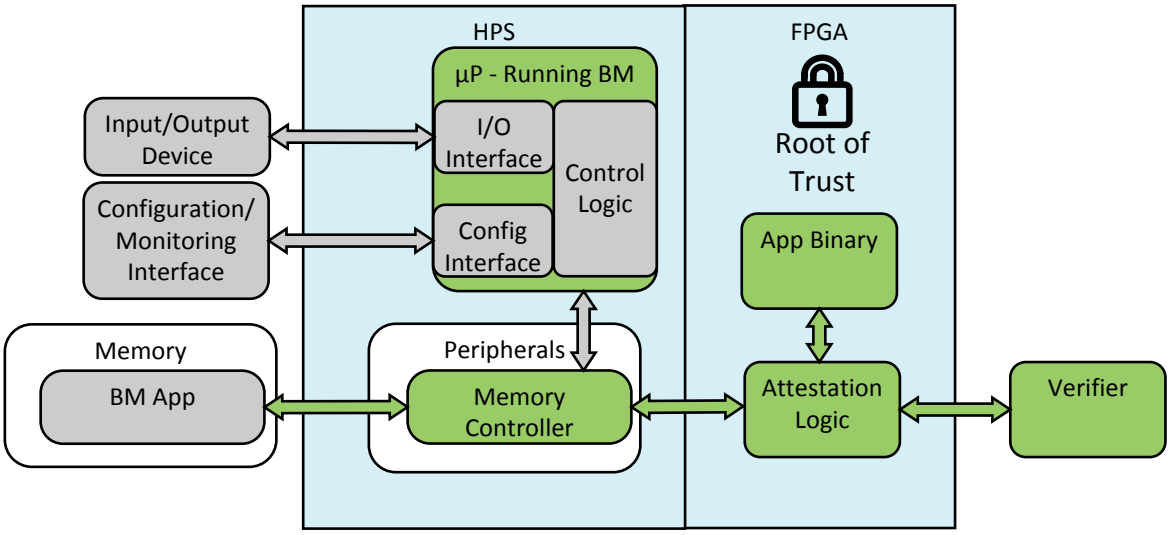

Figure 34: Alternate Attestation Block Diagram

3.5.3.2. For Increased Communication Security

Several algorithms incorporate key sharing and encryption schemes into the attestation logic to address integrity over insecure channels [54][55]. These algorithms would increase communication time by at least the time required for encryption and decryption (AES takes a minimum of 12 cycles for a block of encryption or decryption). Nevertheless, these features could be incorporated into this framework, allowing for tradeoffs between speed of the attestation algorithm, and required cryptographic security. This option is shown in Figure 35, which now includes encrypted communication between the attester and the verifier.

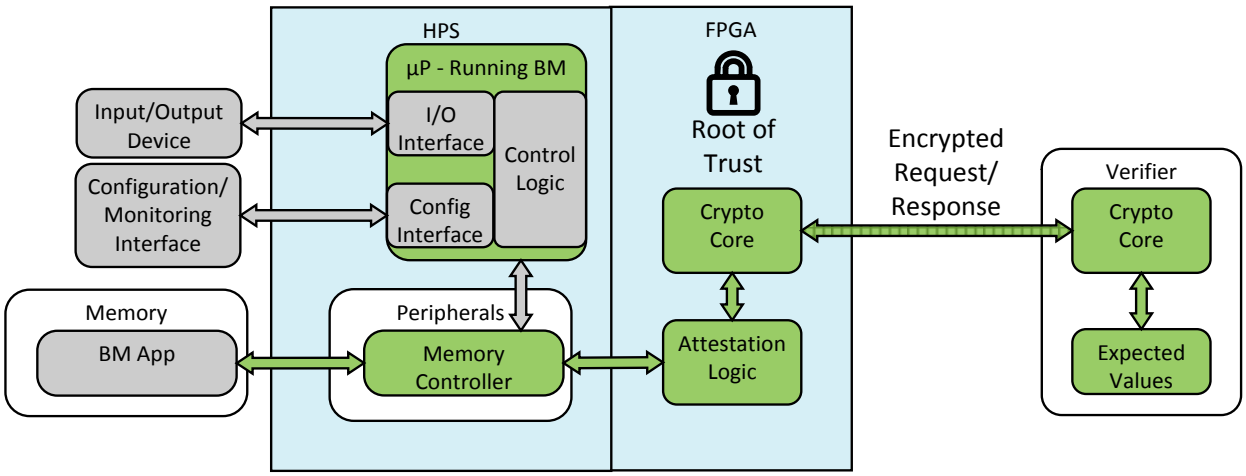

Figure 35: External Verifier with Encryption

3.5.3.3. For Additional Memory And Dynamic Memory Attestation

As an increased convenience, incorporating encryption functionality into the FPGA allows for attestation requests to be passed in-band, through the processor and decrypted in the FPGA, as in [55]. This simplifies physical system construction, with an architecture shown in Figure 36. Note that incorporating the HPS as a proxy for attestation traffic requires tighter timing constraints. The HPS in this case is alerted to the beginning of attestation, and could potentially employ compression or memory copy attacks before transferring the traffic to the trusted FPGA [41]. Additional experimentation would be required to confirm a constant response time of this implementation. This architecture also requires encrypted and authenticated traffic between the FPGA and the verifier.

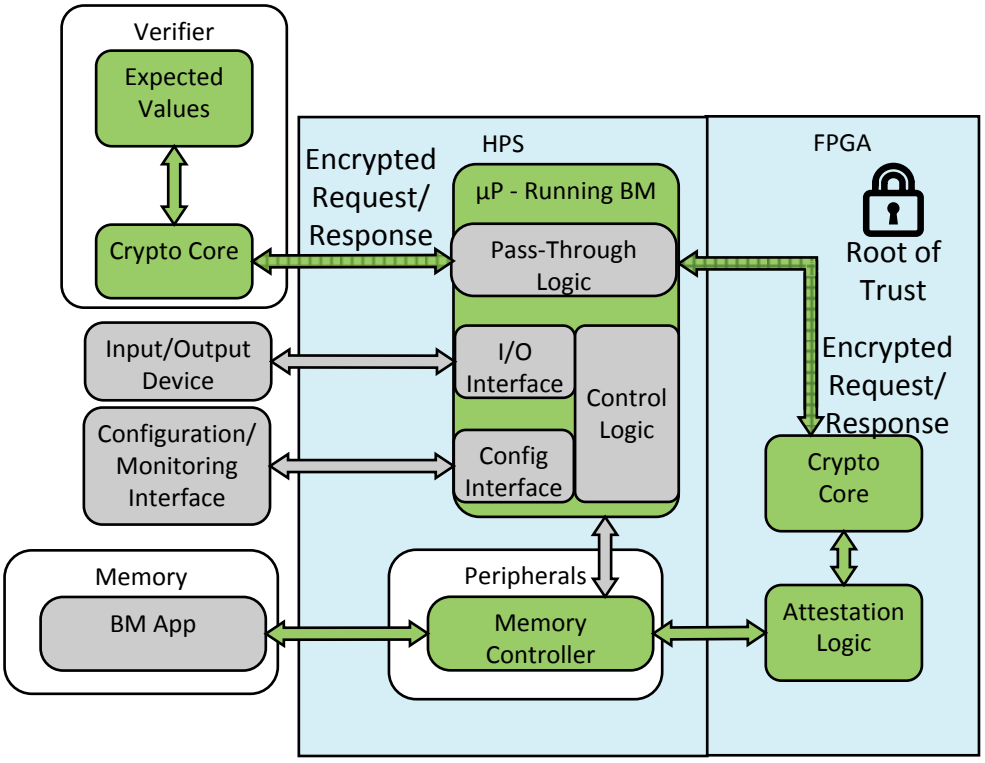

Figure 36: Attestation Architecture Without Physical Security

One limitation to each of these alterations is the lack of introspection into processor registers, required for common execution state attestation algorithms [44], [74]. Gaining ability to incorporate the program and stack pointer into the checksum computation allows for attestation of dynamic properties which solves several potential attestation weaknesses [75]. A solution for incorporating this functionality into the SoC architecture uses the FPGA for local attestation, requesting values, through the processor core in use and verifying the results. This allows for access to processor registers, while limiting variance in timing due to network delay. In addition, this could be paired with any solution previously presented. For example, the FPGA could simply request values from the processor, and run an execution state algorithm in hardware, or run the algorithm entirely in the processor. This allows for the direct inclusion of previously developed software attestation algorithms, while including a hardware root of trust in

the FPGA, similar to [10]. This architecture, shown in Figure 37, is addressed further in Section 7.4.1.1.

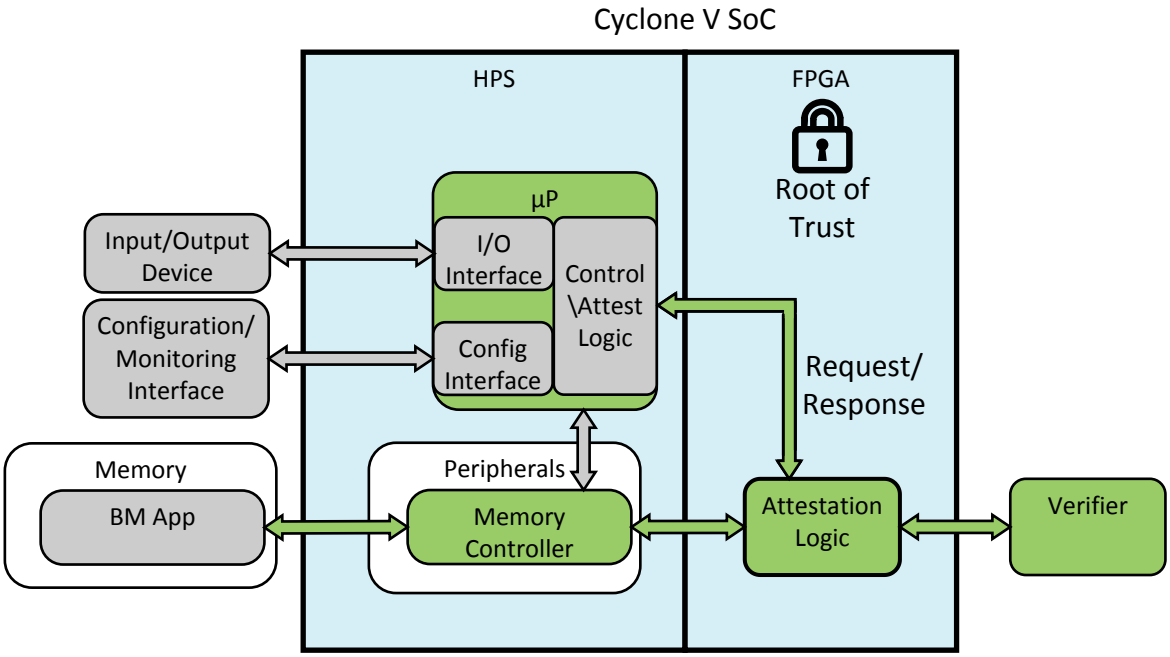

Figure 37: FPGA-Verified Software Attestation

Each of these alternate methods of attestations aid in the removal of the assumption of physical and communication security. It is important to note that using an architecture which does not make these assumptions has a few implications, stated in the following paragraphs.

Timing requirements commonly used in software attestation no longer provide valuable feedback on systems with an untrusted FPGA. The parallelism which drives designers to FPGAs negates any realistic analysis based on time. For example, [8] proposes any delay in runtime signifies malicious activity. An FPGA can redirect memory requests in the same time it takes to service memory requests. Timing bounds serve no purpose in these architectures, other than to serve as a timeout to reattempt transmission.

As a result, architectures which lack physical security rely on the FPGA as a root of trust [32], [55]. This assumption is based on the difficulty of unencrypted bitstream reverse

engineering. While typically assumed to be a non-trivial task, research conducted in this area [23], [76]–[79] proves the potential possibility of this threat. Comparisons of implemented options for attestation is addressed in Section 6.4.

3.5.4. Avalon Bus Communication

The SDRAM bridge chosen for use in Section 3.4.5 may use AXI or Avalon communications. The solution presented utilizes the Avalon-Memory Mapped (MM) bus, primarily for its simplicity [80]. The FPGA in this design masters transactions on the Avalon-MM bus, reading from the memory controller slave. The Avalon-MM bus is synchronous with clock and reset signals, with additional signals for reading and writing data from slave and to slave, respectively. The solution presented only utilizes the Avalon-MM read functionality for attestation. Figure 38 shows an example of an Avalon-MM read and write transfer.

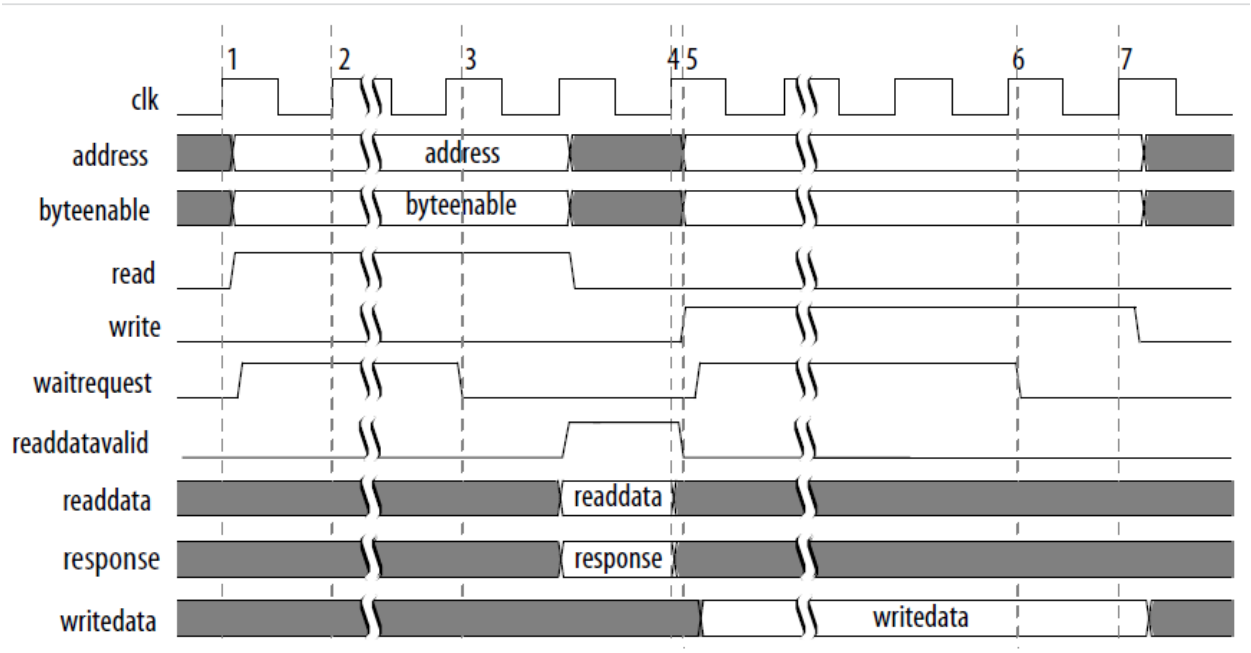

Figure 38: Avalon-MM Read Transaction [80]

VHDL code enables the functionality to request the address value and register the readdata value from the memory controller. The VHDL code must implement a state machine as shown below to master Avalon-MM read transactions. To incorporate bursting requires counters to monitor the current transfer and move to the Init state when the last transfer is received.

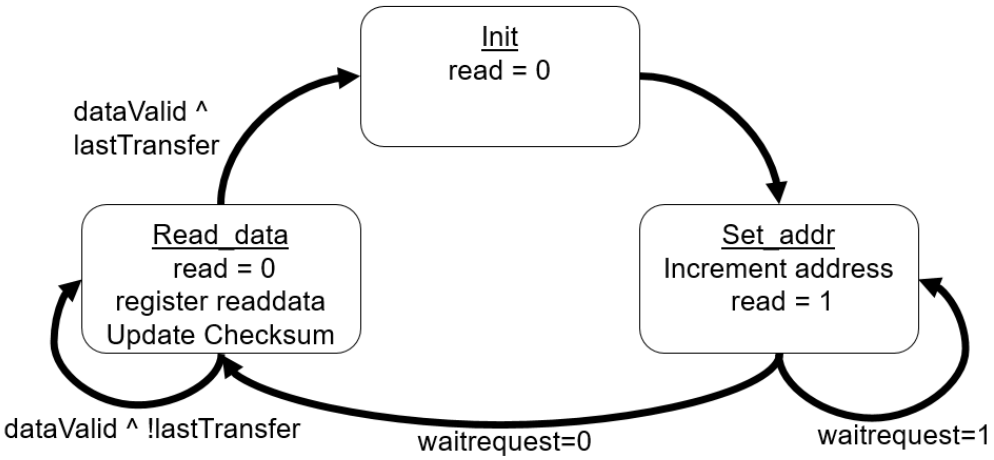

Figure 39: Avalon-MM Master Read State Machine

3.6. Attestation Taxonomy Extension and Additions

The taxonomy in [48], described in Section 2.4, may be applied to the attestation solutions presented in this Chapter. Several fields are added to the taxonomy to further characterize and distinguish between solutions.

First, a field is added for communication security, belonging to either confidential, authentic, both, or neither. Confidential indicates that the message between the verifier and the prover is encrypted. Authentic indicates use of protections against message modification (e.g. message authentication codes). Multiple algorithms require communication security as confidential or authentic, but provide no implementation in their solution.

Second, a field is added for checksum type, belonging to either proven or custom. Proven indicates that the attestation checksum is computed from a proven or well documented

and researched protocol (MD5, SHA, etc.). Custom indicates that the attestation checksum is computed from a custom-made algorithm, proposed along with the attestation framework. For example, SWATT and ICE presented in Chapter II use custom algorithms, while SHA2 is a proven algorithm. Often, the goal of custom algorithms is to reduce computation time with a fully optimized implementation of a sequential algorithm. The goal of proven algorithms is to utilize a trusted one-way function with ideal collision resistance.

Third, processor action is specified, belonging to either attest, transfer or function. Attest refers to the typical software attestation procedure of freezing the processor to execute only attestation code. This often involves disabling interrupts and executing only the attestation code. Transfer refers to the processor receiving an attestation command, and forwarding the message to another entity which completes the attestation. After forwarding the message, the processor continues normal functionality while attestation occurs in the coprocessor. Function refers to the processor always continuing normal operations while attestation is performed, a capability leveraged in SoC attestation solutions. In short, this category captures if the device being attested is the same device performing the attestation.

Finally, a classification of memory traversal is added. The inline class of algorithms do not scan the memory space of the processor, rather they monitor memory accesses either on the bus or as a bump-in-the-wire device.

Table 4 shows the extended taxonomy, applied to attestation algorithms cited in this thesis, as well as new attestation algorithms presented in this thesis. NA refers to a field that is not specified, which may be combined with other options according to application requirements (i.e. some solutions do not require an external verifier and have no mention of communication security). The added fields allow the extended taxonomy to more appropriately fit additional

software integrity schemes, where the original had been tailored to a specific application of software attestation.

Table 4: Attestation Taxonomy Extension

Approach	Evidence	Integrity Measure	Timing	Memory Traversal	Attestation Routine	Program Memory	Data Memory	Interation Pattern	Comm. Sec.	Checksum Type	Proc. Action
Reflection (00)	Software	Static	Loose	Sequential	Embedded	Unfilled	Unverified	One-to-one	None	Proven	Attest
SWATT (04)	Software	Static	Strict	CB-PR	Embedded	Unfilled	Unverified	One-to-one	None	Custom	Attest
ICE (04)	Hybrid	Dynamic	Strict	CB-PR	Embedded	Unfilled	Unverified	One-to-one	None	Custom	Attest
PIV (05)	Software	Dynamic	Loose	Sequential	On-the-fly	Unfilled	Erased	One-to-one	Both	Proven	Attest
Pioneer (05)	Hybrid	Static	Strict	CB-PR	Embedded	Unfilled	Unverified	One-to-one	None	Proven	Attest
SAFE-OPS (05)	Hybrid	Dynamic	Loose	Inline	Embedded	Unfilled	Unverified	One-to-one	N/A	Custom	Function
CODESSEAL (05)	Hybrid	Dynamic	Loose	Inline	Embedded	Unfilled	Unverified	One-to-one	N/A	Custom	Function
M-TREE (06)	Hybrid	Dynamic	Loose	Inline	Embedded	Unfilled	Unverified	One-to-one	N/A	Custom	Function
High-Perf (06)	Hybrid	Dynamic	Loose	Inline	Embedded	Unfilled	Unverified	One-to-one	N/A	Custom	Function
SCUBA (06)	Hybrid	Dynamic	Strict	CB-PR	Embedded	Unfilled	Unverified	One-to-one	None	Custom	Attest
Proactive (07)	Software	Static	Loose	Sequential	Embedded	Filled	Unverified	One-to-one	Conf.	Proven	Attest
Distributed (07)	Software	Static	Loose	BB-PR	Embedded	Filled	Unverified	Many-to-1	Both	Custom	Attest
SAKE (08)	Hybrid	Dynamic	Strict	CB-PR	Embedded	Unfilled	Unverified	One-to-one	Both	Proven	Attest
ReDas (09)	Hardware	Dynamic	Loose	Sequential	Embedded	Unfilled	Verified	One-to-one	None	Custom	Function
MobHat (09)	Hybrid	Static	Loose	Sequential	Embedded	Unfilled	Unverified	One-to-one	Conf.	Proven	Transfer
DataGuard (10)	Software	Dyanmic	Loose	Sequential	Embedded	Unfilled	Verified	One-to-one	None	Proven	Attest
SMARTIES (10)	Hybrid	Dynamic	Loose	Sequential	Embedded	Filled	Erased	One-to-one	None	Proven	Attest
FPGA-Based (11)	Hybrid	Static	Loose	Sequential	Embedded	Unfilled	Unverified	One-to-one	Conf	Proven	Transfer
SMART (12)	Hybrid	Dynamic	Loose	Sequential	Embedded	Unfilled	Verified	One-to-one	None	Proven	Attest
TrustLite (14)	Hybrid	Dynamic	Loose	Sequential	Embedded	Unfilled	Verified	One-to-one	N/A	N/A	Attest
TRAP (15)	Hardware	Static	Loose	Sequential	Embedded	Unfilled	Unverified	One-to-one	Both	Proven	Attest
SEDA (15)	Hybrid	Static	Loose	Sequential	Embedded	Unfilled	Unverified	1-to-many	Both	Proven	Attest
C-FLAT (16)	Hardware	Dynamic	Loose	Sequential	Embedded	Unfilled	Verified	One-to-one	Both	Proven	Attest
DARPA (16)	Hybrid	Static	Loose	Sequential	Embedded	Unfilled	Erased	1-to-many	Both	Proven	Attest
Proposed Solution (Figure 24)	Hybrid	Static	Loose	Sequential	Embedded	Unfilled	Unverified	One-to-one	None	Custom	Function
Proven Logic (Figure 33/ Figure 34)	Hybrid	Static	Loose	Sequential	Embedded	Unfilled	Unverified	One-to-one	N/A	Proven	Function
With AES (Figure 35)	Hybrid	Static	Loose	Sequential	Embedded	Unfilled	Unverified	One-to-one	Conf.	N/A	Function
HPS Proxy (Figure 36)	Hybrid	Static	Loose	Sequential	Embedded	Unfilled	Unverified	One-to-one	Both	Proven	Transfer
FPGA Verified Software-Based (Figure 37)	Hybrid	Static	Loose	Sequential	Embedded	Unfilled	Unverified	One-to-one	Both	Custom	Attest

IV. Experimentation Methodology

This chapter explains the methodology used to characterize the following two effects of attestation:

- the impact of attestation on the HPS response time, and
- the speed of the attestation.

Section 4.1 covers the goals of the experiments conducted and the hypothesis to be proven. Section 4.2 details the system and component under test. Sections 4.3, 4.4, and 4.5 present the experimental metrics, parameters and factors, respectively. Sections 0 and 4.7 detail the two experiments conducted and describe the methodology for analysis of results.

4.1. Goals and Hypothesis

The following experiments quantify the two characterizations above by:

- evaluating the delay introduced into the processor due to attestation, and
- measuring the speed at which the attestation reads memory from the HPS.

Given these results, a designer knows how much the given algorithm will affect the processor response time, and can calculate a priori how long the attestation should take to detect a modification for a given attestation algorithm. An attestation framework which fails to quickly read values, or which introduces unacceptable delay is unlikely to be utilized. The hypothesis is that the FPGA is able to read memory from the HPS at greater than one Gigabyte per second (GB/s) while maintaining a negligible impact on processor response time (<10%). Successfully meeting these numbers would present significant improvement over comparable attestation solutions.

4.2. System Boundaries

The System Under Test (SUT) consists of the ARROW SoCkit Rev. C board [81], shown in Figure 17. The Component Under Test (CUT) consists of the Cyclone V 5CSXFC6D6F31C6 chip. The chip includes the Cyclone V FPGA fabric, as well as the HPS, which includes the Arm-A9 MCU, which includes timers and UART modules required for experimentation. Figure 40 shows the block diagram of the SUT.

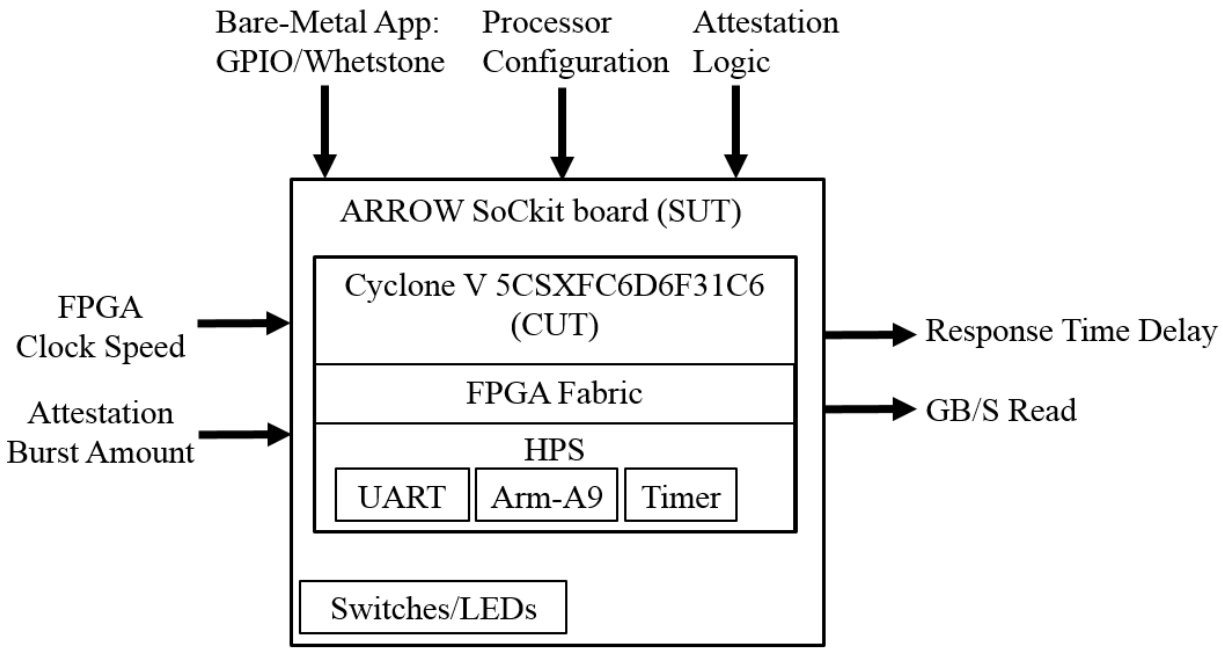

Figure 40: SUT Block Diagram

4.3. Performance Metrics

Metrics recorded for the SUT are degradation in response time of the application running on the HPS, and the speed of memory reading on the FPGA. These metrics characterize the tradeoff between increased rate of detection, and decreased processor throughput.

The first metric, response time degradation, is measured by reading the SP1 timer value in the HPS. This timer runs at 100MHz, creating a 10ns difference between values. The timer is

read before entering the application loop, and upon exiting. Finally, the value is printed to the console through UART, and logged for analysis. Figure 41 shows the pseudocode run to measure this metric.

Compute Processor Delay

```
1: initializeTimer(SP1)
2: initialize(switches)
3: initialize(LEDs)
4: while (1) do
5:     startval ← readTimer(SP1)
6:     switchVal ← readSwitches
7:     computeWhetstone()
8:     setLEDs(switchVal)
9:     endVal ← readTimer(SP1)
10:    print(end − start)
```

Figure 41: Processor Delay Pseudocode

The second metric, FPGA read speed, is obtained through SignalTap, the Altera tool used to introspect and analyze FPGA signals. Through this tool, designers can inspect signals inside the FPGA. The GB/s metric is obtained by counting the clock cycles required to read a set number of address values, and multiplying by the known clock speed. Figure 42 shows a sample output from SignalTap, which shows that the FPGA requests address 0x0010BF40 at clock cycle 16183.

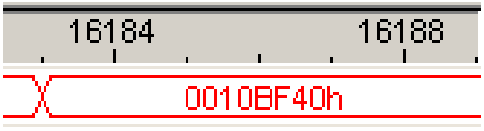

Figure 42: SignalTap Output to Measure Read Speed

4.4. System Parameters

The following parameters affect system performance:

4.4.1. Processor Configuration

The processor configuration, defined in the preloader, affects the clock, memory, interconnects, and other aspects of the HPS. The processor configuration for this example is shown in Figure 43.

```
U-Boot SPL 2013.01.01 (Feb 27 2015 - 08:12:46)
BOARD : Altera SOCFPGA Cyclone V Board
CLOCK: EOSC1 clock 25000 KHz
CLOCK: EOSC2 clock 25000 KHz
CLOCK: F2S_SDR_REF clock 0 KHz
CLOCK: F2S_PER_REF clock 0 KHz
CLOCK: MPU clock 925 MHz
CLOCK: DDR clock  clock 100000 KHz
CLOCK: MMC clock 50000 KHz
CLOCK: QSPI clock 370000 KHz
```

Figure 43: Arm Configuration Parameters

In addition to the parameters shown, the bare metal application described is the only functionality which runs on the processor, running on a single core, compiled without optimization and linked using the Altera toolchain as shown below.

```
$ make
arm-altera-eabi-gcc -g -O0 -Werror -std=c99 -mcpu=cortex-a9 -mfloat-abi=softfp -
mfpu=neon -IC:/altera/14.1/embedded/ip/altera/hps/altera_hps/hwlib/include -c wh
etstone.c -o whetstone.o
arm-altera-eabi-gcc -g -O0 -Werror -std=c99 -mcpu=cortex-a9 -mfloat-abi=softfp -
mfpu=neon -IC:/altera/14.1/embedded/ip/altera/hps/altera_hps/hwlib/include -c io
.c -o io.o
arm-altera-eabi-gcc -g -O0 -Werror -std=c99 -mcpu=cortex-a9 -mfloat-abi=softfp -
mfpu=neon -IC:/altera/14.1/embedded/ip/altera/hps/altera_hps/hwlib/include -c al
t_cache.c -o alt_cache.o
arm-altera-eabi-g++ -TcycloneV-dk-ram-modified.ld -mcpu=cortex-a9 -mfloat-abi=so
ftfp -mfpu=neon -IC:/altera/14.1/embedded/ip/altera/hps/altera_hps/hwlib/include
 -L./alt_cache.o whetstone.o io.o alt_cache.o -o whetstone.axf
arm-altera-eabi-objcopy -O binary whetstone.axf whetstone.bin
mkimage -A arm -T standalone -C none -a 0x100040 -e 0 -n "baremetal image" -d wh
etstone.bin whetstone-mkimage.bin
Image Name:   baremetal image
Created:      Fri Oct 21 09:35:56 2016
Image Type:   ARM Linux Standalone Program (uncompressed)
Data Size:    61496 Bytes = 60.05 kB = 0.06 MB
Load Address: 00100040
Entry Point:  00000000
```

Figure 44: Compilation of Bare Metal App

Also, to reduce jitter and variance and create the worst-case processor delay, caches are disabled. The 32Kb L1 instruction and data caches and the 512Kb shared cache would reduce the degradation of the processor response time, and disabling these features gives the worst-case for attestation impact. Enabling these caches for realistic designs presents a potential security challenge depending on the allocation, replacement, and write policies of the caches. This topic is out of the scope of this research.

4.4.2. Attestation Logic and Configuration

Any change in the attestation logic coded in VHDL and synthesized in the FPGA would likely affect the operation of the attestation. The only change in the FPGA functionality comes from altering the clock speed and burst amount requested, addressed in Section 4.5. This also includes keeping constant the choice of using the Avalon bus to communicate with the memory controller over 256-bit width, as specified in Section 3.4.5.

In addition, to simulate an ideal communication protocol, the external verifier in these tests resides inside the FPGA. An additional module communicates with the attestation logic, creating a minimum of communication delay between the external verifier and the attestation algorithm.

4.4.3. Application Parameters

While results are normalized against the default of no attestation, any change in the functionality or time required to run the application may impact results. All experiments run the same application on the HPS, which completes the Whetstone benchmark 10 times with a loop value of 20.

4.5. Factors

The following factors are varied in order to characterize the attestation. Table 5 shows the factors and levels used for both experiments.

4.5.1. FPGA Clock Speed

The clock speed of the FPGA affects the speed of attestation and the speed of requests sent to the memory controller. This has no effect on the clock used for the microprocessor or any HPS peripherals. The levels for clock speed in MHz are shown below. Speeds in excess of 200MHz may produce erratic behavior.

4.5.2. FPGA Burst Amount

The burst amount defines the number of sequential, uninterrupted responses expected once the memory controller chooses to satisfy the burst. The five levels are shown.

Table 5: Experiment 1 (Response Time Degradation) and experiment 2 (Memory Read Speed) Factors and Levels

Factor	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6	Level 7	Level 8	Level 9	Level 10
FPGA Clock Speed (MHz)	20	40	60	80	100	120	140	160	180	200
Burst Amt	0	63	127	191	255					

4.6. Evaluation Technique

4.6.1. Experiment 1 – Processor Response Time Degradation

The purpose of experiment 1 is to determine the degradation in response time of the processor application as a result of running attestation through the memory control shown in Figure 28. The application runs through approximately 144,000 floating point calculation loops for each time measurement. Each time measurements is taken 40 times to provide small confidence intervals.

This research hypothesizes that increases in the burst amount of the attestation causes the greatest effect on the processor runtime, due to the fact that the memory controller does not interrupt a burst once beginning to service it. Likewise, increases in FPGA clock speed may result in proportional increases in processor runtime. This research may also find an optimal minimum, where decreasing the FPGA clock speed and maintaining bursts further increases processor runtime, due to the same uninterruptable nature of bursts. At very low speeds, non-bursting attestation may be recommended. If these results maintain a predictable curve across various applications, a designer can deterministically choose how to implement attestation to continue to meet timing requirements with acceptable margins.

4.6.2. Experiment 2 – FPGA Memory Read Speed

The purpose of experiment 2 is to determine the speed at which the FPGA reads memory addresses given various configurations of attestation. The amount of clock cycles to read through approximately 40,000 addresses will be measured 10 times.

This research hypothesizes that memory read speed follows a relatively predictable path, where faster clock speeds result in linearly faster read speeds, as well as higher bursts resulting in faster speeds. However, a maximum will likely be found where increasing FPGA speeds no longer increases read speed linearly. Having knowledge of both the influence of clock speed on processor response time, and the resulting speed at which the application is read allows for a decision of the optimal attestation speed to run at for an intended application.

4.7. Experimental Design

Experiment 1 is a full factorial design and uses all factors described. The total number of trials is therefore 10 FPGA clock speeds * 5 FPGA burst amounts * 40 measurements = 2000. Experiment 2 is a full factorial design with a total number of 500 trials. For both experiments, a one variable t-test is performed on each experimental scenario to determine the mean processing delay, standard deviation, standard error, and 99% confidence interval.

V. Experimental Results and Analysis

5.1. Timing Experiments

Chapter IV introduced experiments to characterize the attestation implemented. The following sections address the results of these experiments.

5.1.1. Processor Delay Time

Processor delays were measured as described in Chapter IV and plotted in Figure 45. Every line represents a different FPGA clock speed, ranging from 20-200MHz over 10 different speeds. The five values on the x-axis represent the different burst amounts. A burst value of 0 or 1 means that the FPGA requests one address and gets one response. A burst value of 255 means the FPGA requests one address with a burst response and receives 255 responses to complete the transfer. The y-axis represents the additional percent delay, normalized against the time for no attestation occurring. All data points include error bars representing a 99% confidence interval on the mean.

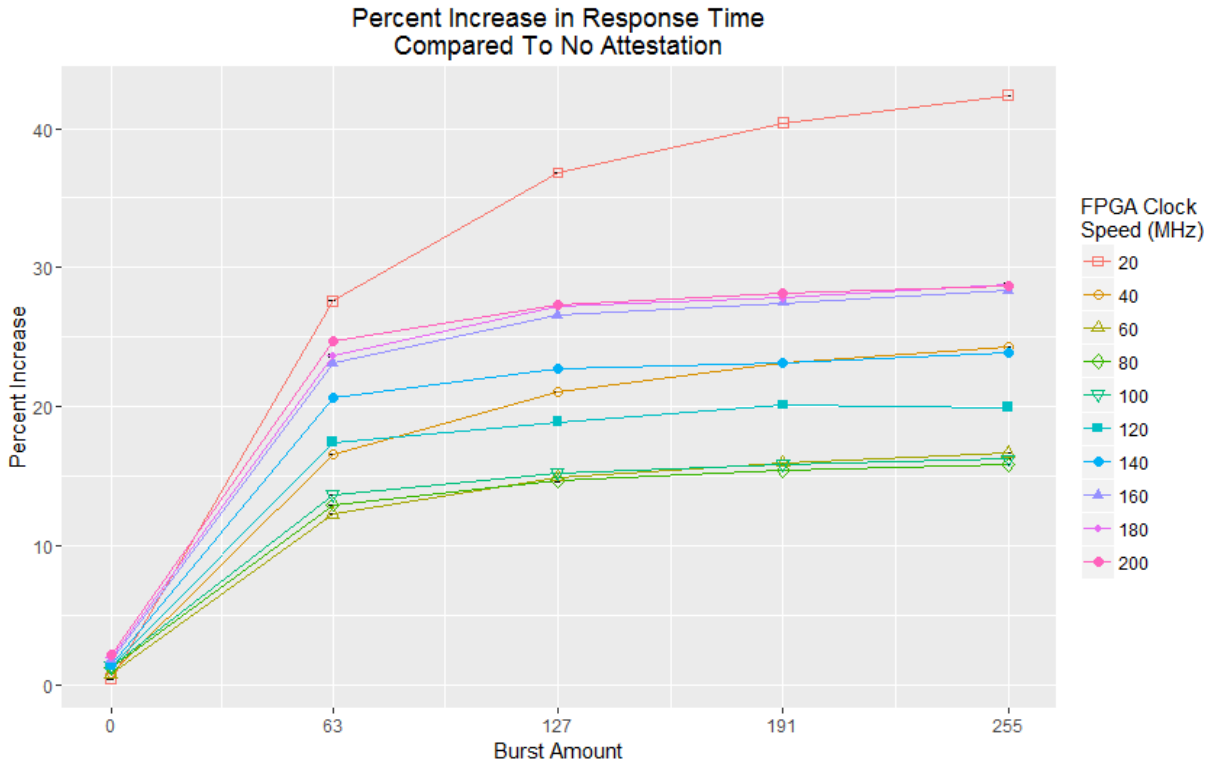

Figure 45: Measurement of Processor Delay

For detailed results, refer to Appendix A, which tabulates the results of the one-variable t-test performed on the FPGA clock speed factor with a burst amount of 0, 63, 127, 191, and 255 respectively.

The near-linear increase in delay across clock speeds without bursting is expected (Figure 46). As more requests come in, the likelihood of stalling the processor increases. The average increase in delay as a result of incorporating bursting is also expected. Once the memory controller begins a burst, the transfer continues until completed. This creates a long delay where the processor waits for instructions to execute.

An interesting side-effect of bursting is the non-monotonically increasing effect on response time due to increased FPGA speed. During a burst, neither a relatively fast or slow FPGA speed impacts the processor less. The lowest FPGA speed created the worst delay; the highest FPGA speed created the second worst delay. A median speed of 60-80MHz creates the

lowest effect (Figure 47). The determination of the precise relationships which cause this effect is out of scope of this research.

Figure 46 and Figure 47 show these results, comparing the FPGA speed against the response time delay for a specific burst amount during attestation, along with 99% confidence intervals.

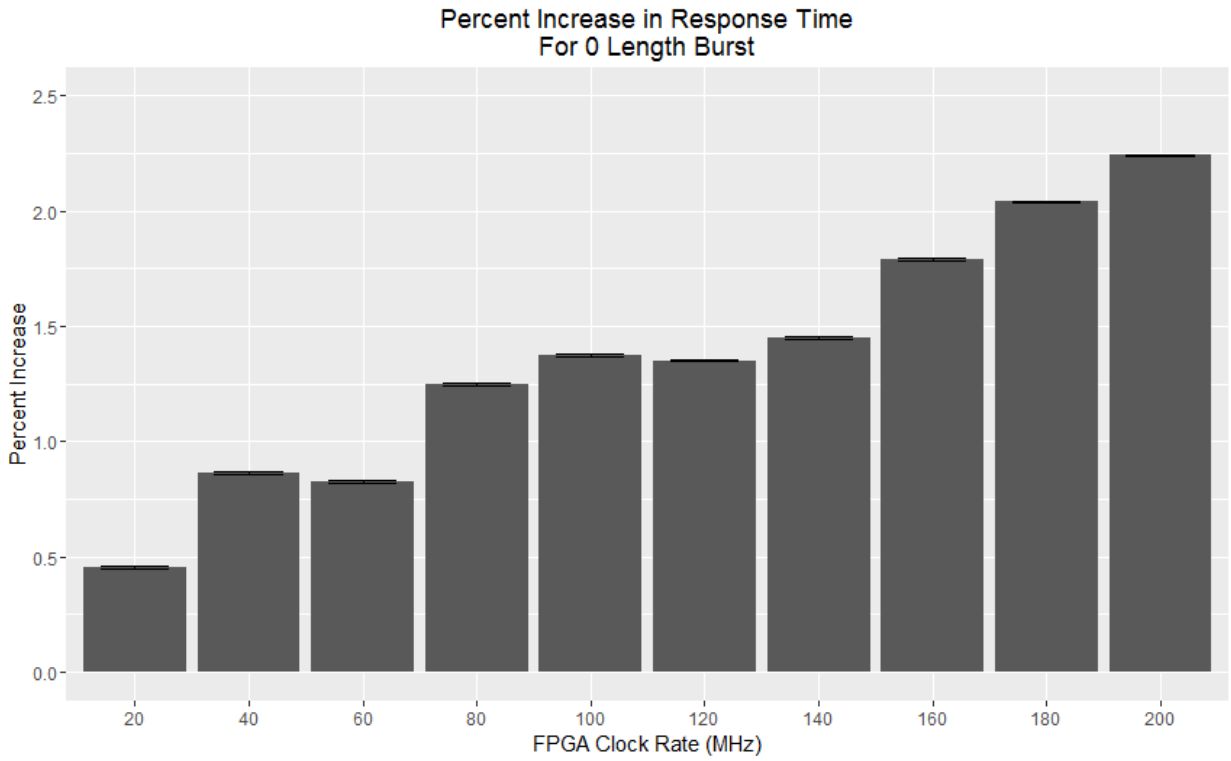

Figure 46: Processor Delay for 0 Length Bursts

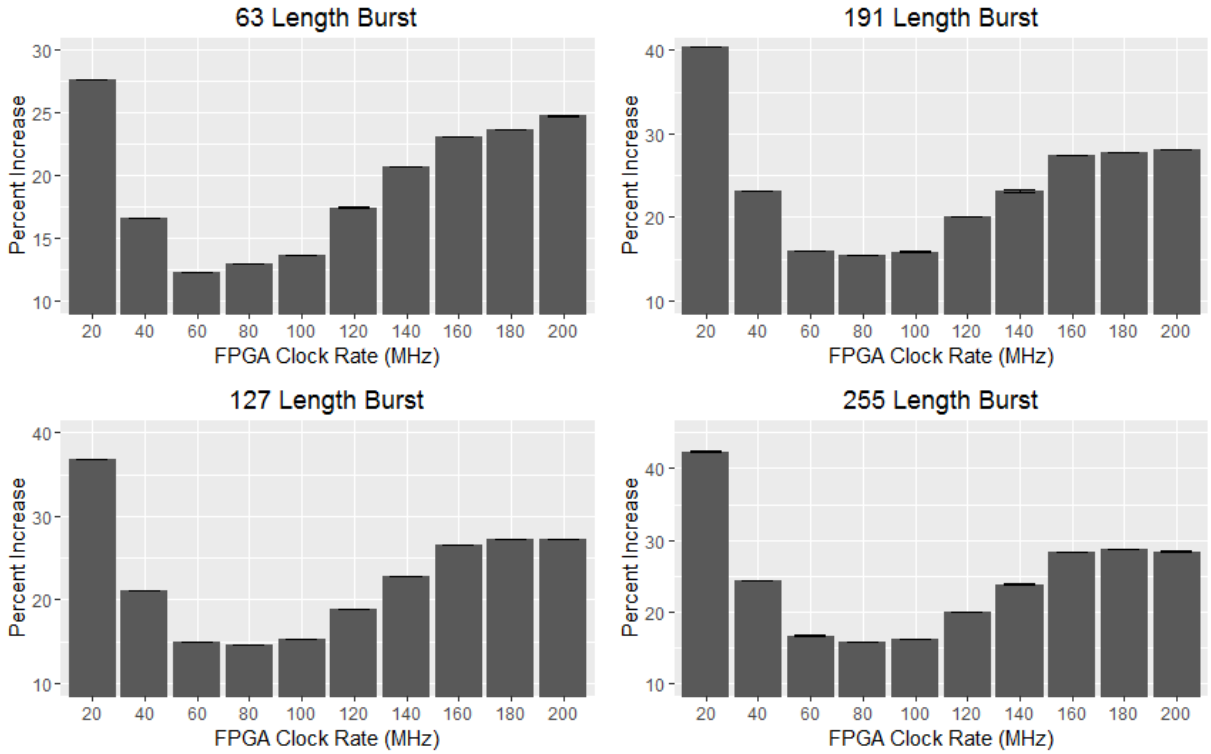

Figure 47: Processor Delay for 63, 127, 191, and 255 Length Bursts

5.1.2. Memory Read Speed

Processor delays were measured as described and plotted in Figure 48. Varying lines and the x-axis represent the same factors as mentioned above, and the y-axis represents the number of Gigabytes read by the FPGA per second. All data points include error bars representing a 99% confidence interval on the mean.

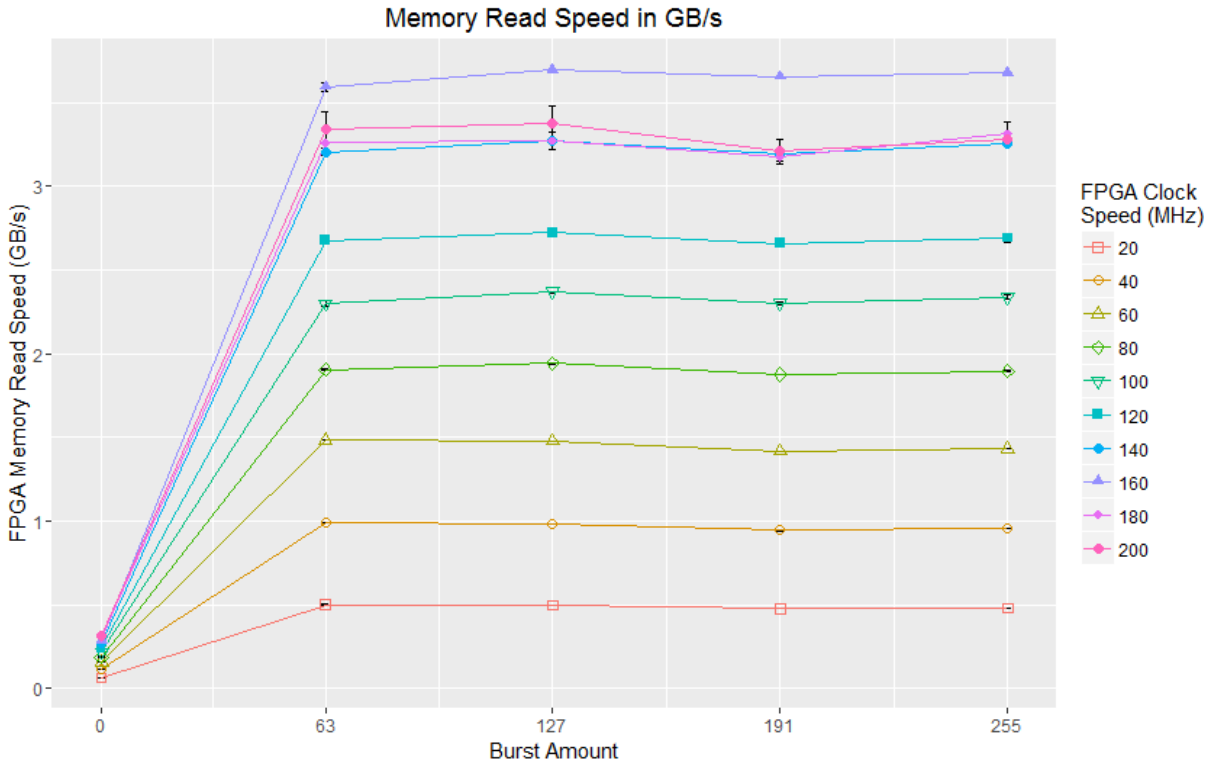

Figure 48: Memory Read Speed Results

Refer to Appendix A for results of the one-variable t-test performed on the FPGA clock speed factor with a burst amount of 0, 63, 127, 191, and 255 respectively. The mean values show the GB/s of memory read from the FPGA, giving a representation of potential attestation speed.

The results for read speed for each burst amount are shown in Figure 49 and Figure 50. As expected, the increase in FPGA clock speed results in a nearly linear increased memory read speed (Figure 49), and the incorporation of bursting increases read speed as well. In addition, once passing a certain clock speed while utilizing bursts, the read speed declines (Figure 50). This may indicate that a quality of service metric in the memory controller limits the response time in order to reduce further impacting the operation of the processor.

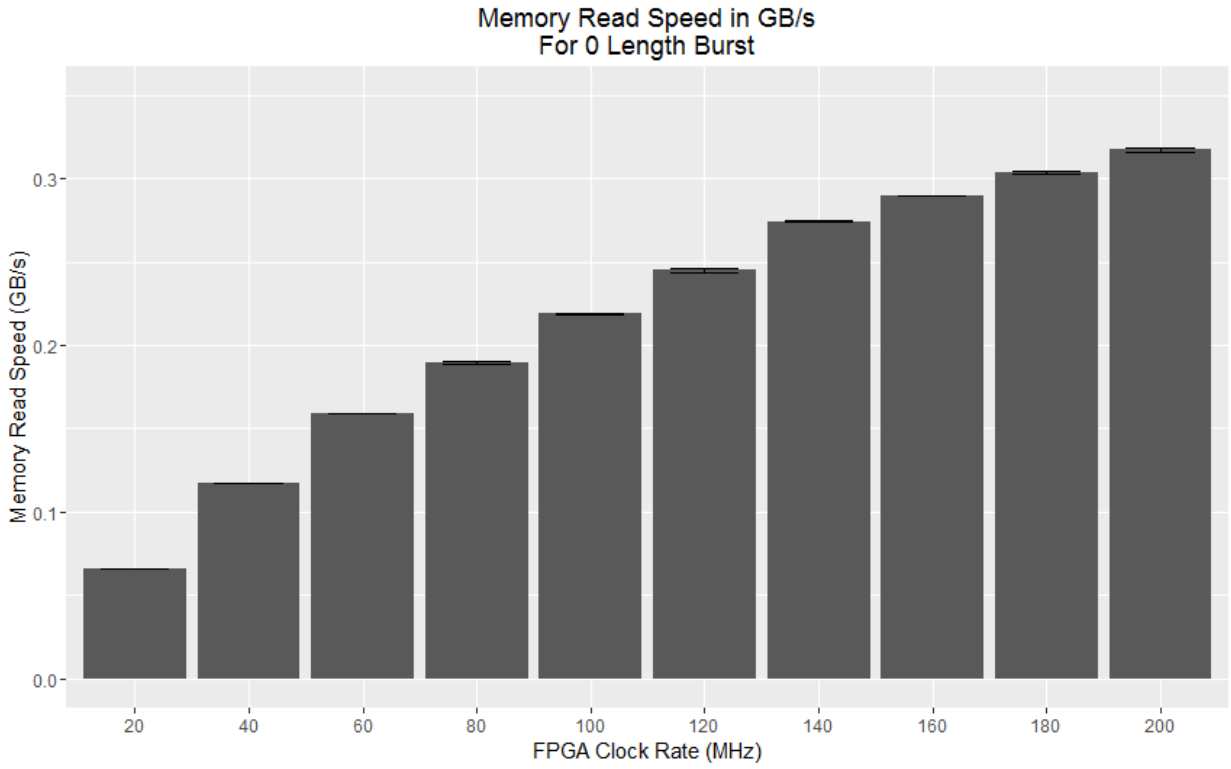

Figure 49: Memory Read Speed for 0 Bursts

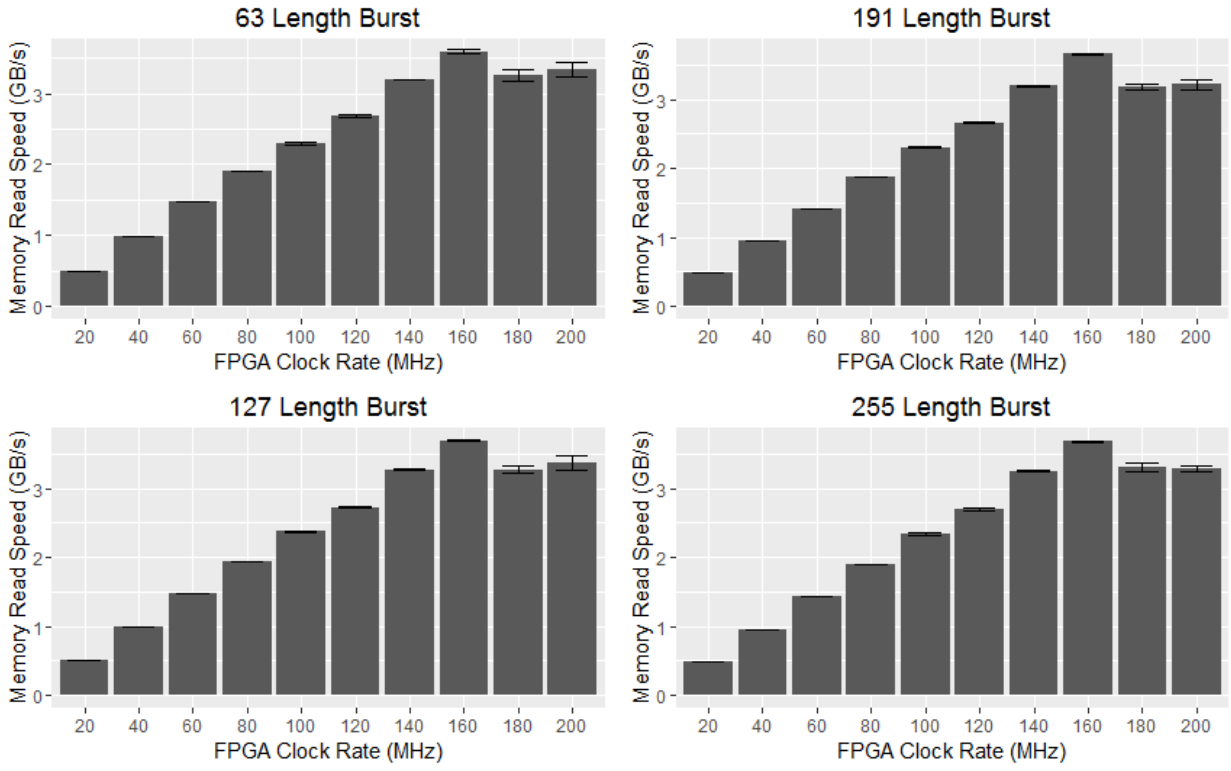

Figure 50: Memory Read Speed for 63, 127, 191, and 256 Bursts

5.2. Overall Analysis

The experiments show the wide range of effects and speeds that attestation can take for a given algorithm. Due to the optimized speed of the algorithm chosen, these results show the capability of attestation for maximum processor effect and memory read speed. Given the range of choices for burst amount and FPGA clock speed, no configuration satisfies the hypothesis for this experiment (above 1GB/s with under 10% processor impact). In order to achieve the negligible impact on processor runtime, non-bursting logic is required. Making this decision then reduces the read speed to less than 0.5GB/s, below the goal of 1GB/s. Both experiments show that the decision to utilize bursting or not creates a large impact on the attestation speed and processor effect. For all configurations, the hypothesis is not met, and if this optimized algorithm fails to meet this goal, any reduction in speed (additional cryptographic primitives, communication protocols, etc.) will further fail to meet the goal.

VI. Architectural Analysis

6.1. Attestation Implementation

The attestation algorithm described in Section 3.5.1 was implemented on a single Cyclone V SoC, and the external verifier was implemented on a Raspberry Pi microcontroller, communicating over UART, shown in Figure 51. The Raspberry Pi runs a python script to request attestation checksums from the SoC and reports the results to a console. In response to threats, a designer may choose to have the verifier alert additional modules to the change or reset the SoC. The response to detection of threats is out of the scope of this research.

Figure 51: Experimental Setup

6.2. Verify Detection of Modifications

6.2.1. Simulated Threat

The threat simulated in this section is a code injection attack, proven feasible in Harvard architectures to achieve arbitrary code execution [9]. A demonstration of exploiting a vulnerability to inject code into an application is outside of the scope of this research. As a

substitute, an FPGA module injects malicious functionality into the application while the attestation algorithm also runs. Figure 52 shows the block diagram for this simulated threat.

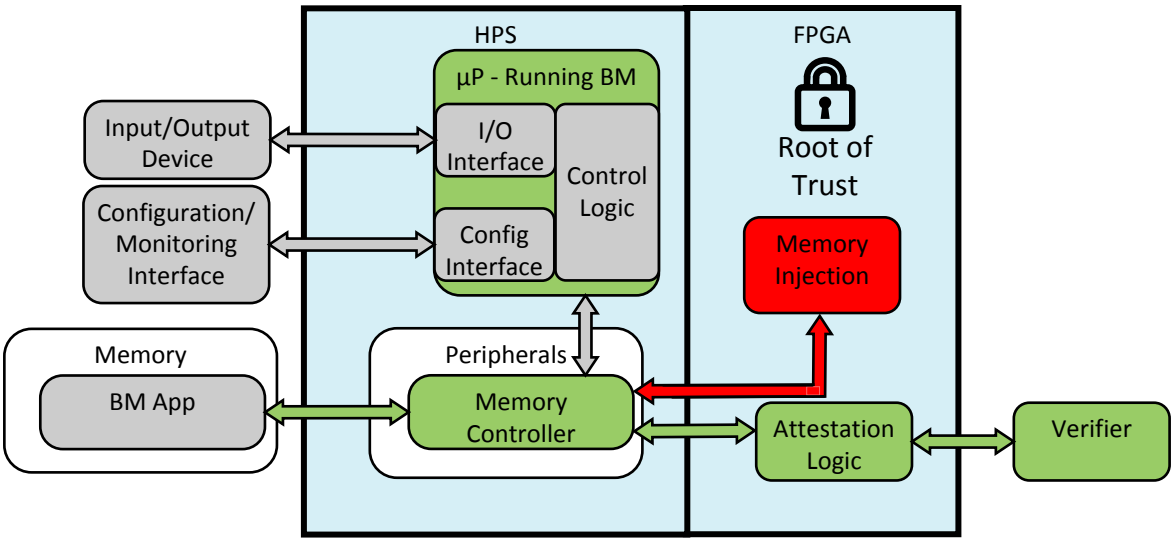

Figure 52: Simulated Threat Block Diagram

After implementing attestation and before error injection, the external verifier confirms in Figure 53 that the application is trusted.

```
pi@raspberrypi:~/Desktop $ python attest.py

Correct Response
```

Figure 53: External Verifier Trusted Output

When the memory injection module activates, a single byte in the application memory is overwritten. Figure 54 shows a portion of the device controller application hex. The memory injection module overwrites the address at offset EA4 in the application with a value of '140053E3'. This overwrites an index in a large 'for' loop, causing a significant increase in the runtime of the program. This simple example demonstrates the potentially harmful effects of a small code injection attack.

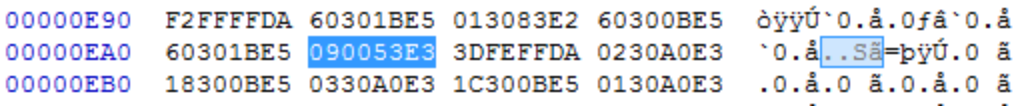

Figure 54: Application Hex for Error Injection

6.2.2. Error Detection

When the attestation logic computes and returns an incorrect checksum, the external verifier reports the status. In this situation, where the output of the application appears unaltered, a user would otherwise have no feedback about the corrupted state of the application. Attestation enables this new capability, shown in Figure 55.

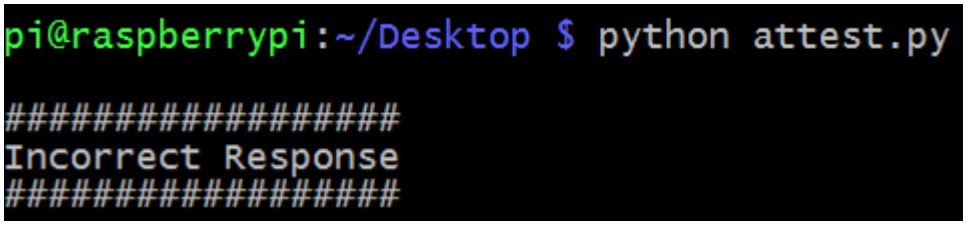

Figure 55: External Verifier Error Output

6.3. Update Procedure

As described in Section 3.1.2.2, the entire system utilizing the SoC must support reprograming. This section describes the methodology used to reconfigure the functionality of the HPS given the solution described in Section 3.4.1.

First, the user makes the necessary changes to the application. After compiling the application with the appropriate commands, the user has a new binary image to load onto the SD card for the device controller. The compilation stage is shown in Figure 56.

```
$ make
arm-altera-eabi-gcc -g -O0 -Werror -std=c99 -mcpu=cortex-a9 -mfloat-abi=softfp -
mfpu=neon -IC:/altera/14.1/embedded/ip/altera/hps/altera_hps/hwlib/include -c wh
etstone.c -o whetstone.o
arm-altera-eabi-gcc -g -O0 -Werror -std=c99 -mcpu=cortex-a9 -mfloat-abi=softfp -
mfpu=neon -IC:/altera/14.1/embedded/ip/altera/hps/altera_hps/hwlib/include -c io
.c -o io.o
arm-altera-eabi-gcc -g -O0 -Werror -std=c99 -mcpu=cortex-a9 -mfloat-abi=softfp -
mfpu=neon -IC:/altera/14.1/embedded/ip/altera/hps/altera_hps/hwlib/include -c al
t_cache.c -o alt_cache.o
arm-altera-eabi-g++ -TcycloneV-dk-ram-modified.ld -mcpu=cortex-a9 -mfloat-abi=so
ftfp -mfpu=neon -IC:/altera/14.1/embedded/ip/altera/hps/altera_hps/hwlib/include
 -L./alt_cache.o whetstone.o io.o alt_cache.o -o whetstone.axf
arm-altera-eabi-objcopy -O binary whetstone.axf whetstone.bin
mkimage -A arm -T standalone -C none -a 0x100040 -e 0 -n "baremetal image" -d wh
etstone.bin whetstone-mkimage.bin
Image Name:   baremetal image
Created:      Fri Oct 21 09:35:56 2016
Image Type:   ARM Linux Standalone Program (uncompressed)
Data Size:    61496 Bytes = 60.05 kB = 0.06 MB
Load Address: 00100040
Entry Point:  00000000
```

Figure 56: Image Compilation

The user then takes this golden copy, and obtains the expected checksum for given input values. The verifier is updated to contain these new checksum values and program address space. After programming the FPGA, the verifier requests attestation checksums and compares the result with expected values. This process may be repeated as often as necessary by designers.

6.4. Comparison of Implemented Alternate Options

The algorithm described in Section 3.5.3 represents a relatively simple and fast choice for implementing software attestation. As an alternative, additional choices were proposed, implemented, and are discussed further here.

6.4.1. Increased Checksum Security through Secure Hash Algorithm

Created in 2001 by the National Security Agency, SHA-256 remains in use today as a means to create a secure hash of an arbitrary length of input data. As a potential alternative to the algorithm presented above, this hashing algorithm could attest to the state of the memory,

where a single byte change would perturb the entire resulting SHA-256 output. This improves security by using a tested security algorithm instead of the more custom logic proposed above. This also severely reduces attestation speed, where every 512 bit chunk of data must be run through a 64 round checksum.

The implementation was constructed as described in Section 3.5.3.1 and shown in Figure 33**,** with the attestation logic using the algorithm shown in Figure 57 below. Note that this algorithm assumes a 256-bit bus, and an increment in address index 'i' results in a 256-bit address increment. A smaller bus results in additional requests for each 512-bit chunk required by SHA-256.

The algorithm starts as before, with the address range, starting address, and initial checksums as inputs. The logic requests two 256-bit responses from the memory controller, and feeds them into a SHA-256 core to update the digest value. After reading all memory values, the checksum is returned to the verifier.

Compute Attestation With SHA256

INPUT: Address range A, Starting address A_0, Initial Checksums C_1, C_0
OUTPUT: Final Checksum C

1: **for** $A_i \in A$ **do** //Even i, read linearly, starting at A_0
2: $SHA\_IN \leftarrow memoryVal(A_i)|memoryVal(A_{i+1})$
3: $digest \leftarrow SHA256(digest, SHA\_IN)$
4: $i+ = 2$
5: **return** $digest$

Figure 57: Attestation Algorithm with SHA256

There are three implications of choosing this algorithm over the logic presented in Section 3.5.1. These relate to the speed, security, and area of the final application.

First, SHA-256 requires 64 rounds of computation per input chunk. This means that for every 512 bit input, the hash calculation blocks the attestation. This will result in a roughly 30x

increase in the time required for attestation, requiring a SHA update every two memory requests. Preliminary experiments with a two-request burst shown above results in the expected 30x decrease in read speed at a test of 100 MHz FPGA clock speed. While this standard sequential hash computation significantly degrades attestation speed, the potential does exist to utilize hash trees and parallelize hundreds of SHA hash cores to reduce the impact on speed [82].

Second, the security of the checksum logic increased in this application. Often motivated by size, weight, and power requirements, embedded applications opt to utilize unproven cryptographic implementations [83]. As a result, exploiting collisions allows an attacker to compromise the system. In this case, the proven SHA hashing algorithm allows for increased security, if a significant reduction in speed is acceptable.

Third, to implement hashing functionality requires additional utilization of FPGA fabric. In the case of the Cyclone V, this addition is trivial. A single SHA-256 core requires less than 1 percent of the Cyclone V FPGA fabric.

6.4.2. Increased Communication Security Through Encryption

Shown in Figure 35, encryption can be used to secure the communication between the external verifier and the attestation logic. In an application without physical security, this addition would become mandatory. The Advanced Encryption Standard (AES), established by NIST in 2001 provides symmetric encryption of arbitrary length data with a block size of 128 bits and key sizes of 128,192 or 256 bits. AES encryption in Electronic Code Book (ECB) mode is used here to encrypt communications between the verifier and the attestation logic. ECB operates on data chunks independently, making it easily parallelizable in FPGAs. As a security

consequence, ECB is deterministic, and often not recommended. The attestation logic implementing AES-ECB encryption is shown below.

First, the request is received and broken into chunks to be decrypted in parallel. After decrypting, the starting address, address range, and initial checksums are extracted from the user defined mapping of the chunks. Then the user defined attestation algorithm is performed, this may be the SHA algorithm, or the fast algorithm presented in Chapter III. Finally, the cleartext checksum is encrypted and sent to the verifier.

Algorithm: COMPUTE ATTESTATION WITH AEC-ECB

INPUT: AES-ECB encrypted checksum containing address range A, starting address A_0, initial checksum C_0

OUTPUT: Encrypted Checksum C_e

1: $request \leftarrow receive(attestationRequest)$
2: $attestationChunks[] \leftarrow breakIntoChunks(request)$
3: $decryptedChunks \leftarrow map(decryptECB, attestationChunks)$
4: $\{A, A_0, C_0\} \leftarrow decryptedChunks$ //user defined mapping
5: $C \leftarrow attest(A, A_0, C_0)$ //user defined attestation algorithm
6: $C_e \leftarrow map(encryptECB, C)$
7: **return** C_e

Figure 58: Attestation Logic With AES-ECB

The same implications can be made on the attestation as before. For speed, ECB encryption/decryption can operate on a 128-bit block of data and output data in 12 clock cycles, resulting in a minimal effect compared to the thousands of cycles required to fully compute attestation in Step 5.

Encryption of traffic results in increased security in two applications: when attestation requests are forwarded through the processor, as shown in Figure 36, or physical security is not assumed. If either of these cases are true, encryption should be used. Additional space required for each 128 chunk ECB encryption and decryption module is less than 1 percent of the Cyclone

V FPGA fabric. Choosing to instead use CBC mode of AES also requires less than 1 percent of the Cyclone V FPGA fabric, though it would utilize additional time for both encryption and decryption depending directly on the length of the message transmitted.

6.5. Additional Threats Mitigated

In addition to the memory injection attack described above, the architecture presented allows for the defense against additional threats which commonly reduce the efficacy of software attestation frameworks.

A number of attestation threats result from the fact that the processor has knowledge of the attestation request, and uses this as trigger for actions to overcome the attestation. The framework presented defends against these attacks, commonly referred to as Time of Check to Time of Use (ToCToU) [48] attacks by doing the following two things:

- routing attestation requests directly through the FPGA, and
- removing the ability for the processor to monitor memory requests.

Exploits triggered by the start of attestation include compression, proxy, collusion, and precomputation attacks [48]. Given that the processor has no ability to determine when attestation takes place, these attacks no longer present a threat. Thus, this framework allows for defense against potentially the largest issue which plagues traditional software attestation applications [48][41].

Assuming trusted communication between the verifier and the trusted FPGA on the device defends against replay and impersonation attacks. There are no potential malicious actors attached to the communication medium which can attempt to impersonate the device or replay

responses and trick the verifier. Additional threats which are not mitigated by this framework are detailed in the following section.

6.6. Threats Not Mitigated

While defending against a large number of threats, the framework presented does fail to secure against several threats which maintain the original application binary. First, ROP attacks do not alert attestation. This attack uses small sequences of instructions found in the application, chained together with commands which alter program flow (return, branch, etc.) to execute arbitrarily complex functionality. ROP attacks maintain the original instruction code, meaning this attack successfully bypasses static attestation techniques. Likewise, memory copy attacks maintain original instruction code, and are undetected by static attestation. Memory copy attacks create an altered set of instructions, allowing attestation to continue on the original application.

6.7. Limitations

6.7.1. FPGA Write Access to HPS

This solution requires the assumption of the trusted programmer. Indeed, this requirement represents a single point of failure for the system. An adversary who alters the FPGA bitstream may arbitrarily overwrite processor instruction memory, a capability leveraged for experimentation. In fact, TrustZone support in the Cyclone V only allows for disabling of both read and write. No means exist to disable FPGA writes while also allowing reads, as long as the preloader and FPGA bitstream allows both [17]. An architecture which allows for further control, and read-only access to certain regions of memory would increase resistance to this threat.

6.7.2. Unattestable Memory Spaces

As mentioned in Section 3.4.5, utilizing only the FPGA2SDRAM bridge means the FPGA has no access to certain regions of memory available to the processor. Some regions have little impact on security, and map to registers which have no functional impact on the system related to attestation. Other regions create difficulties for attestation. For example, 64KB of on-chip RAM available to the processor does not map to SDRAM, and is not accessible behind the FPGA2SDRAM bridge. The capability to attest to this region requires enabling and utilizing the FPGA2HPS bridge, relying on HPS responses, and implementing a software-attestation based approach, as in [8]. This alternative is presented in Section 3.5.3.

6.7.3. Modified Harvard Architecture with Absence of MPU

The Arm-A9 chip, used in the Cyclone V, has a modified Harvard architecture. This means that while separate pathways exist for data and instruction fetches, the same memory space stores both these types of fields. As a result, an adversary may utilize a software vulnerability such as a buffer overflow to place and execute code in the process stack to gain control of the system. A memory protection unit (MPU) allows for the marking of address spaces as non-executable. This feature, available in ARM Cortex-M and R series processors, allows a designer to implement data execution prevention, a common security tool in many commercial operating systems. The Cyclone V chip does not contain an MPU, and as a result cannot prevent against stack or heap execution. For this reason, a final solution utilizing this architecture should be built on a different SoC, or incorporate execution state attestation. The first option is available in SoCs from Microsemi and Xilinx, and the second is addressed in Section 7.4.2.1.

6.7.4. Potentially Overoptimistic Assumptions

Noted in [48], there is a gap between the assumptions and threat models of many attestation proposals, and the operational environments and realistic capabilities of an adversary. Threats continue to exist even after attestation assumes them away. This section revisits the assumptions presented and analyses either ways to potentially further mitigate the threats negated by their presumption, or argues for their realism.

6.7.4.1. Physical and Communication Security.

With the potential attacks carried out by malicious actors with physical access, and the ease at which physical security controls are often broken [84], assuming away physical security presents a dangerous and often exploited single point of failure for any security system. This assumption is overoptimistic, and realistic security controls should be taken regardless of attempts at physical security. Fortunately, FPGAs offer the hardware required to secure both the physical components and the communication channels used [27], [32], [39], [85].

6.7.4.2. Hardware Trust

Utilizing malicious hardware negates any security controls in place. The potential exists for a wide range of trojans to be placed in hardware to execute various attacks against the system [86], [87]. This assumption is realistic if designers rely on trusted foundries and suppliers for hardware, through programs established by DARPA and the NSA [68]. There also exists the potential to identify malicious functionality by analysis techniques including RF fingerprinting [88].

6.7.4.3. Trusted Verifier

Utilizing the two assumptions above allows for a verifier to be implemented on trusted hardware in a system which allows for physical security properties. The trusted verifier, implemented on an FPGA, may be secured from malicious activity by the same properties that allow for the FPGA to act as a root of trust in the proposed framework, and this assumption is realistic.

6.7.4.4. Trusted Programmer and Golden Copy

The threat model in Section 3.2 presents a scenario where an adversary compromises the system, premised on the fact that networked systems become compromised. This did not extend to the compromise of the trusted programmer, which may additionally store the golden copy of the application. Placing these systems on a separate, isolated network only reduces the potential of their exploitation. Considering an air-gap may simply act as a low-latency connection, isolation does not fully remove this threat, and this assumption may be overoptimistic [89]. Additional steps should be taken to secure the systems which generate the hardware images, and compile and store the applications.

VII. Conclusions

This chapter summarizes the goals (Section 7.1), conclusions (Section 7.2), contributions (Section 7.3) and future work (Section 7.4) of the research presented, and concludes with final recommendations (Section 7.5).

7.1. Motivation and Research Goals

Software attestation aims to provide trust in applications running on untrusted systems. Previous research in this field relied on a wide array of hardware, methodologies, algorithms, and assumptions to accomplish their goal [48]. Hardware-based software attestation often relies on custom modules and tamper-resistant hardware which complicates development [90], [91]. FPGA-based software attestation research is limited in scope, and SoC architectures have yet to be leveraged for attestation [54], [55]. The goal of this research is to analyze the potential use of SoC architectures for software attestation. In order to accomplish this, several stages of research were completed.

7.2. Conclusions

The research conclusions presented in each chapter are addressed here. First, Chapter III presented tradeoffs made in the design of the framework. Each choice is made with the goal of creating a fast, simple-to-use, trustworthy attestation architecture. While one architecture is chosen for implementation, the variety of decisions made during this stage presents a wide array of potential solutions meeting different objectives with significant tradeoffs. Several of these potential alternatives are also addressed in this chapter, with additional options implemented and described in Chapter VI.

To characterize the impact of attestation on processor functionality, two experiments are performed. The first experiment shows that in the worst case, attestation creates a 42% increase in processor loop time and in the best case, a less than 1% increase. The second experiment shows that read times of attestation vary from 0.06 GB/s to 3.2 GB/s. These results show that at every data point, the read speed and processor delay fail to meet the hypothesized 1 GB/s at under a 10% impact. This impact may meet the real-time requirement for some applications, and fail for others. Care should be used when incorporating this attestation into time-sensitive applications.

For the attestation functionality analysis, two simple experiments are completed. First, a simulated memory injection attack is detected by the attestation algorithm, and displayed by the verifier. Second, a run of the update procedure demonstrates compliance with the reconfigurable requirement of the framework. These experiments show the attestation detects the attack presented in the threat model, and enables designers to customize the attestation framework to their applications.

7.3. Contributions

The prevalence of embedded systems which comprise sensitive markets mandates methods to ensure reliability and trust in embedded software. Current research in software attestation aims to accomplish this goal. As FPGAs expand in use, attestation solutions which leverage their capabilities present unique opportunities for security without costly, custom-hardware configurations. This research presented the first attempt at adapting a COTS SoC chip for direct application of software attestation. The simplicity of the framework allows for easy adaptation, while also allowing for a hardware root of trust to secure a system. The portability of

the framework enables attestation of any application which may be compiled and run on the ARM MPU in use. The security of the framework creates a second line of defense against threats which aim to compromise embedded systems.

7.4. Future Work

The exploratory nature of this research revealed significant potential for future development. The following sections detail additional work which could potentially benefit attestation on SoC systems, organized into self-contained research ventures for parallelized and rapid development.

7.4.1. Integrate into Unsecured Environment

Research similar to this thesis is presented in [54], [55]. In these publications, the authors focus more heavily on securing communications between the verifier and the hardware monitor, while using the untrusted processor as a proxy. The principles and communication security presented in their research, matched with the SoC architecture presented here, could create a secure system without the need for custom bus connections [54] or soft processors [55]. Implementation of their security research would allow for effective removal of the most overoptimistic assumption presented in this thesis.

7.4.1.1. Analyze Attestation Opportunities In Soft-Core Processors

The authors in [54] utilized a soft-core processor as a method to simulate an FPGA with DMA to processor memory on a connected bus. The use of a soft-core processor allows for additional features which could be exploited for attestation in ways that a SoC does not allow.

For example, a soft-core processor could be coded in order to allow for runtime hardware access to registers, or creation of execute-only memory. To-date, no attestation solution utilizes a soft-core processor to allow for these benefits.

7.4.2. Expand Attestation

7.4.2.1. Incorporate Execution State Attestation

As noted in [92], [93], many current software attestation architectures and algorithms (including the one presented here) focus on analysis of the static portions of the system memory. Attacks that violate dynamic system properties go undetected. To address this threat, execution state attestation aims to attest to the running state of the program in its dynamic context. Several attestation algorithms aim to incorporate dynamic context into the computation by including system registers including the program counter, and data or instruction pointer [42], [44], [45], [74]. Additional solutions use data flow graphs, control flow integrity analysis, and data guards to attest to the dynamic state of the execution [91], [93], [94]. Defenses specifically targeted at ROP attacks include the use of execute-only memory, a method that has not yet been combined with any attestation approach [48]. Any combination of these techniques could be potentially incorporated into the framework provided, porting the attestation solutions presented in past literature into the configuration shown in Figure 37.

7.4.2.2. Reach Entire Memory Space

As noted in Section 3.5.2, various regions of memory including HPS on-chip RAM are inaccessible to FPGA requests. As a result, the 64kb of RAM could be used to store malicious instructions in a space which cannot be attested given the current framework. Figure 37 presents

additional functionality to resolve this limitation. Future work may pair the algorithm presented for quick attestation of the larger SDRAM address space, while implementing a traditional software-based attestation approach to request attestation of the smaller on-chip RAM space through the processor. Pairing this with a dynamic attestation solution may allow for a full defense against threats to any memory space.

7.4.3. Expand to Additional Architectures and Boards

The Cyclone V SoC contains several limitations which prevent successful implementation of desired functionality, including the lack of non-executable memory space. SoC architectures from Xilinx and Microsemi contain microcontrollers which solve this problem. Porting this algorithm to various SoC chips may reveal additional strengths and weaknesses of platforms from the various SoC manufacturers.

7.4.4. Attest Operating System Application

As noted in Section 3.4.6, bare metal programming provides reduced runtimes while also decreasing jitter for an application when compared to running on an operating system. Unfortunately, bare metal programming also reduces portability, and increases development times for a given functionality. Research presents a means to implement attestation to an application running on an operating system [54], [55]. This research could be applied to this architecture to allow for faster software development times, increasing ease of use.

Attestation of embedded operating system applications itself may present additional research possibilities, as typical software approaches are often targeted at highly resource constrained environments with static memory [48].

7.4.5. Send Dynamic FPGA Updates to Incorporate Changes to Attestation

As noted in the taxonomy in Table 4, the attestation presented in this research utilizes an embedded attestation algorithm. The dynamic update capability of FPGAs could allow for this algorithm to be updated on-the-fly, reducing the capability for an adversary to exploit the algorithm, or reverse engineer the configuration bitstream.

7.5. Concluding Thoughts and Recommendations

This research proves the potential feasibility of utilizing SoCs for attestation. Despite this capability, the solution contains serious limitations. Before deployment of attestation solutions built on SoC architectures, several aspects of the future work presented must be addressed. This research presents a first step towards the use of SoCs for attestation; development is still required to deliver a solution which adequately protects high-value assets against advanced threats.

Appendix A. Experimental Results Tables

Table 6: 0 Burst, Processor Degradation Results

Clock Speed	% Increase From No Attestation	Samples	Standard Deviation	Standard Error of the Mean	Confidence Interval (99%)
20	0.452911	40	0.004865	0.000769	0.001981
40	0.862058	40	0.005088	0.000804	0.002072
60	0.823362	40	0.006346	0.001003	0.002585
80	1.248348	40	0.004194	0.000663	0.001708
100	1.372873	40	0.004711	0.000745	0.001919
120	1.351199	40	0.004471	0.000707	0.001821
140	1.449117	40	0.005378	0.000850	0.002190
160	1.791671	40	0.006156	0.000973	0.002507
180	2.040399	40	0.005053	0.000799	0.002058
200	2.241151	40	0.006093	0.000963	0.002482

Table 7: 63 Burst, Processor Degradation Results

Clock Speed	% Increase From No Attestation	Samples	Standard Deviation	Standard Error of the Mean	Confidence Interval (99%)
20	27.61260	40	0.008726	0.001380	0.003554
40	16.63696	40	0.008534	0.001349	0.003476
60	12.32006	40	0.010875	0.001720	0.004429
80	12.96827	40	0.006419	0.001015	0.002614
100	13.67651	40	0.006943	0.001098	0.002828
120	17.45746	40	0.006161	0.000974	0.002509
140	20.65990	40	0.007419	0.001173	0.003022
160	23.13067	40	0.008334	0.001318	0.003394
180	23.66619	40	0.006559	0.001037	0.002671
200	24.75327	40	0.006346	0.001003	0.002584

Table 8: 127 Burst, Processor Degradation Results

Clock Speed	% Increase From No Attestation	Samples	Standard Deviation	Standard Error of the Mean	Confidence Interval (99%)
20	36.82964	40	0.011159	0.001764	0.004545
40	21.08058	40	0.010769	0.001703	0.004386
60	14.93724	40	0.011660	0.001844	0.004749
80	14.66177	40	0.007599	0.001201	0.003095
100	15.26089	40	0.007526	0.001190	0.003065
120	18.90798	40	0.020970	0.003316	0.008540
140	22.74506	40	0.008745	0.001383	0.003562
160	26.60557	40	0.010659	0.001685	0.004341
180	27.27239	40	0.007729	0.001222	0.003148
200	27.28293	40	0.008435	0.001334	0.003435

Table 9: 191 Burst, Processor Degradation Results

Clock Speed	% Increase From No Attestation	Samples	Standard Deviation	Standard Error of the Mean	Confidence Interval (99%)
20	40.38663	40	0.008874	0.001403	0.003614
40	23.13798	40	0.013466	0.002129	0.005484
60	15.99817	40	0.018869	0.002983	0.007685
80	15.42543	40	0.008110	0.001282	0.003303
100	15.89657	40	0.006980	0.001104	0.002843
120	20.13801	40	0.010652	0.001684	0.004338
140	23.14774	40	0.103039	0.016292	0.041965
160	27.46844	40	0.010002	0.001582	0.004074
180	27.83331	40	0.007611	0.001203	0.003100
200	28.12258	40	0.008287	0.001310	0.003375

Table 10: 255 Burst, Processor Degradation Results

Clock Speed	% Increase From No Attestation	Samples	Standard Deviation	Standard Error of the Mean	Confidence Interval (99%)
20	42.38077	40	0.017605	0.002784	0.0071700
40	24.28079	40	0.015807	0.002499	0.006438
60	16.68236	40	0.023430	0.003705	0.009542
80	15.81139	40	0.006252	0.000988	0.002546
100	16.26832	40	0.011967	0.001892	0.004874
120	19.96931	40	0.029801	0.004712	0.012137
140	23.85145	40	0.014138	0.002235	0.005758
160	28.38160	40	0.012386	0.001958	0.005045
180	28.78537	40	0.006660	0.001053	0.002712
200	28.68641	40	0.014826	0.002344	0.006038

Table 11: 0 Burst, Memory Read Speed Results

Clock Speed, 0 Burst	Memory Read Speed	Samples	Standard Deviation	Standard Error of the Mean	Confidence Interval (99%)
20	0.065930106	10	0.000022138	7E-06	1.8E-05
40	0.117242264	10	0.000157066	4.97E-05	0.000128
60	0.158743061	10	0.000083438	264E-05	6.8E-05
80	0.189921657	10	0.000992817	0.000314	0.000809
100	0.218948676	10	0.000762483	0.000241	0.000621
120	0.245055446	10	0.001519639	0.000481	0.001238
140	0.274404237	10	4.48719E-05	1.42E-05	3.66E-05
160	0.289645114	10	0.000112055	3.54E-05	9.13E-05
180	0.303489326	10	0.000810618	0.000256	0.00066
200	0.317121794	10	0.001762709	0.000557	0.001436

Table 12: 63 Burst, Memory Read Speed Results

Clock Speed, 0 Burst	Memory Read Speed	Samples	Standard Deviation	Standard Error of the Mean	Confidence Interval (99%)
20	0.502574	10	0.000482	0.000152	0.000392
40	0.990231	10	0.000675	0.000214	0.000550
60	1.485346	10	0.001013	0.000320	0.000825
80	1.90672	10	0.003817	0.001207	0.003109
100	2.298986	10	0.019351	0.006119	0.015762
120	2.67861	10	0.022287	0.007048	0.018154
140	3.202306	10	0.005980	0.001891	0.004871
160	3.594698	10	0.035059	0.011087	0.028558
180	3.260172	10	0.094645	0.029929	0.077093
200	3.33846	10	0.130407	0.041238	0.106223

Table 13: 127 Burst, Memory Read Speed Results

Clock Speed, 0 Burst	Memory Read Speed	Samples	Standard Deviation	Standard Error of the Mean	Confidence Interval (99%)
20	0.497613	10	0.000246	7.76478E-05	0.0002
40	0.986859	10	0.001285	0.00040633	0.001047
60	1.480288	10	0.001927	0.000609495	0.00157
80	1.943801	10	0.003742	0.001183262	0.003048
100	2.370468	10	0.005346	0.001690445	0.004354
120	2.722062	10	0.01619	0.005119701	0.013187
140	3.273766	10	0.007817	0.002471966	0.006367
160	3.693155	10	0.00989	0.003127525	0.008056
180	3.271805	10	0.064658	0.020446515	0.052667
200	3.374713	10	0.125238	0.039603807	0.102013

Table 14: 191 Burst, Memory Read Speed Results

Clock Speed, 0 Burst	Memory Read Speed	Samples	Standard Deviation	Standard Error of the Mean	Confidence Interval (99%)
20	0.478192	10	0.000279	8.8341E-05	0.000228
40	0.94603	10	0.001574	0.00049785	0.001282
60	1.419044	10	0.002362	0.00074678	0.001924
80	1.875316	10	0.002858	0.00090376	0.002328
100	2.301098	10	0.009849	0.00311454	0.008023
120	2.659302	10	0.012069	0.00381666	0.009831
140	3.192877	10	0.009483	0.00299886	0.007725
160	3.655405	10	0.009403	0.00297336	0.007659
180	3.172814	10	0.051623	0.01632447	0.042049
200	3.215833	10	0.081782	0.02586176	0.066615

Table 15: 255 Burst, Memory Read Speed Results

Clock Speed, 0 Burst	Memory Read Speed	Samples	Standard Deviation	Standard Error of the Mean	Confidence Interval (99%)
20	0.482507	10	0.000231	7.30061E-05	0.000188
40	0.956665	10	0.00124	0.000391986	0.00101
60	1.434998	10	0.001859	0.000587978	0.001515
80	1.897787	10	0.003768	0.001191507	0.003069
100	2.339045	10	0.014343	0.004535613	0.011683
120	2.692481	10	0.02895	0.009154819	0.023581
140	3.251038	10	0.005298	0.001675267	0.004315
160	3.677312	10	0.014682	0.00464283	0.011959
180	3.314779	10	0.081263	0.025697601	0.066193
200	3.284752	10	0.051845	0.016394774	0.04223

Bibliography

[1] INTEL, “Rise of the Embedded Internet,” *Intel white Pap.*, no. January 2009, pp. 1–5, 2009.

[2] F. Neto and P. Neto, *Designing Solutions-Based Ubiquitous and Pervasive Computing: New Issues and Trends*, 1st ed. IGI Global, 2010.

[3] B. Krebs, “Source Code for IoT Botnet ‘Mirai’ Released,” 2016. [Online]. Available: https://krebsonsecurity.com/2016/10/source-code-for-iot-botnet-mirai-released/. [Accessed: 14-Oct-2016].

[4] N. Falliere, L. O. Murchu, and E. Chien, “W32 . Stuxnet Dossier,” vol. 4, no. February, pp. 1–69, 2011.

[5] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, “A Large-Scale Analysis of the Security of Embedded Firmwares,” *USENIX Secur. Symp.*, pp. 95–110, 2014.

[6] J. Zaddach and A. Costin, “Embedded Devices Security and Firmware Reverse Engineering,” *Black Hat USA*, p. 9, 2013.

[7] G. Coker *et al.*, “Principles of remote attestation,” *Int. J. Inf. Secur.*, vol. 10, no. 2, pp. 63–81, 2011.

[8] A. Seshadri, A. Perrig, L. Van Doom, and P. Khosla, “SWATT: SoftWare-based ATTestation for embedded devices,” *Proc. - IEEE Symp. Secur. Priv.*, vol. 2004, pp. 272–282, 2004.

[9] S. I. Cedex, C. Castelluccia, and I. Rhône-alpes, “Code Injection Attacks on Harvard-Architecture Devices Categories and Subject Descriptors.”

[10] H. Tan, W. Hu, and S. Jha, “A TPM-enabled remote attestation protocol (TRAP) in wireless sensor networks,” *Proc. 6th ACM Work. Perform. Monit. Meas. Heterog. Wirel. wired networks - PM2HW2N ’11*, p. 9, 2011.

[11] J. Zambreno, D. Honbo, a. Choudhary, R. Simha, and B. Narahari, “High-Performance Software Protection Using Reconfigurable Architectures,” *Proc. IEEE*, vol. 94, no. 2, pp. 1–13, 2006.

[12] S. M. Trimberger, “Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology,” *Proc. IEEE*, vol. 103, no. 3, pp. 318–331, 2015.

[13] K. Morris, “All is Not SRAM,” *FPGA J.*, pp. 5–9, 2007.

[14] S. M. Trimberger and J. J. Moore, “FPGA security: Motivations, features, and applications,” *Proc. IEEE*, vol. 102, no. 8, pp. 1248–1265, 2014.

[15] S. Drimer, “Volatile FPGA design security – a survey,” *Univ. Cambridge*, pp. 1–51, 2008.

[16] “Zynq ® UltraScale +^{TM} MPSoCs Product Selection Guide,” *Xilinx, Inc.*, 2016.

[17] Altera, “Cyclone V Hard Processor System Technical Reference Manual,” 2015.

[18] ARM, “AMBA AXI and ACE Protocol Specification,” pp. 1–306, 2011.

[19] B. Guide, “HPS SoC Boot Guide - Cyclone V SoC Development Kit Boot Overview,” pp.

1–30, 2016.

[20] Altera, “Booting and Configuration Introduction,” vol. 6.30, 2014.

[21] Xilinx Inc., “Using Encryption to Secure a 7 Series FPGA Advanced Encryption Standard (AES) and Authentication Encrypted Bitstream Implementation Overview,” vol. 1239, pp. 1–15, 2015.

[22] Xilinx Inc., “XAPP1084(v1.3): Developing Tamper Resistant Designs with Xilinx Virtex-6 and 7 Series FPGAs,” *Xilinx, Inc.*, vol. 1084, pp. 1–20, 2013.

[23] R. S. Chakraborty, I. Saha, A. Palchaudhuri, and G. K. Naik, “Hardware trojan insertion by direct modification of FPGA configuration bitstream,” *IEEE Des. Test*, vol. 30, no. 2, pp. 45–54, 2013.

[24] Xilinx Inc., “Secure Boot of Zynq-7000 All Programmable SoC,” vol. 1175, 2013.

[25] Altera, “Cyclone V Device Handbook,” vol. 1, no. January, pp. 1–66, 2014.

[26] S. Feedback, “Altera Remote Update IP Core User Guide Avalon-MM in Altera Remote Update IP Core,” 2016.

[27] M. Barbareschi, E. Battista, A. Mazzeo, and S. Venkatesan, “Advancing WSN physical security adopting TPM-based architectures,” *Proc. 2014 IEEE 15th Int. Conf. Inf. Reuse Integr. IEEE IRI 2014*, pp. 394–399, 2014.

[28] J. Ho, “Distributed Software-Attestation Defense Against Sensor Worm Propagation,” vol. 2015, 2015.

[29] “Evita-Project,” 2011. [Online]. Available: www.evita-project.org. [Accessed: 17-Nov-2016].

[30] M. S. Idrees, H. Schweppe, Y. Roudier, M. Wolf, D. Scheuermann, and O. Henniger, “Secure automotive on-board protocols: A case of over-the-air firmware updates,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 6596 LNCS, pp. 224–238, 2011.

[31] E. Khayari, T. Paristech, and R. B. Gmbh, “Secure Automotive on-Board Electronics Network Architecture,” *FISITA World Automot. Congr. 2010*, no. 1, 2010.

[32] M. Wolf and T. Gendrullis, “Design, Implementation, and evaluation of a vehicular hardware security module,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 7259 LNCS, pp. 302–318, 2012.

[33] A. Seshadri and A. Perrig, “Using Software-based Attestation for Verifying Embedded Systems in Cars,” *Proc. Embed. Secur. Cars Work.*, 2004.

[34] K. Eguro and R. Venkatesan, “FPGAs For Trusted Cloud Computing,” *F. Program. Log. Appl.*, pp. 63–70, 2012.

[35] O. Machidon, F. Sandu, C. Zaharia, P. Cotfas, and D. Cotfas, “Remote SoC / FPGA Platform Configuration for Cloud Applications,” *Optim. Electr. Electron. Equip. (OPTIM), 2014 Int. Conf.*, pp. 827–832, 2014.

[36] N. Jazdi, “Cyber physical systems in the context of Industry 4.0,” *2014 IEEE Autom. Qual. Testing, Robot.*, pp. 2–4, 2014.

[37] N. Roy, “December 2015 Altera Corporation PLC Architecture in the Industry 4.0 World: Challenges, Trends, and Solutions,” *Sr. Strateg. Mark. Manag. – Ind.*, no. December, pp. 1–7, 2015.

[38] A. Perrig and B. Sinopoli, “Mechanisms to Provide Integrity in SCADA and PCS devices,” *Proc. Int. Work. Cyber-Physical Syst. Appl.*, 2008.

[39] M. Barbareschi, E. Battista, V. Casola, A. Mazzeo, and N. Mazzocca, “On the adoption of FPGA for protecting cyber physical infrastructures,” *Proc. - 2013 8th Int. Conf. P2P, Parallel, Grid, Cloud Internet Comput. 3PGCIC 2013*, pp. 430–435, 2013.

[40] Symantec, “Duqu,” *Symantec Secur. Response*, vol. 1.4, pp. 1–71, 2011.

[41] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, “On the difficulty of software-based attestation of embedded devices,” *CCS ’09 Proc. 16th ACM Conf. Comput. Commun. Secur.*, pp. 400–409, 2009.

[42] A. Seshadri, M. Luk, and A. Perrig, “SAKE: Software attestation for key establishment in sensor networks,” *Ad Hoc Networks*, vol. 9, no. 6, pp. 1059–1067, 2011.

[43] IBM, “Introduction To The Citadel Architecture.” p. 63, 1990.

[44] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla, “Using FIRE & ICE for Detecting and Recovering Compromised Nodes in Sensor Networks,” *Program*, no. December, p. 26, 2004.

[45] A. Seshadri, A. Perrig, M. . Luk, L. Van Doom, E. Shi, and P. Khosla, “Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems,” *Oper. Syst. Rev.*, vol. 39, no. 5, pp. 1–16, 2005.

[46] J. Zambreno, A. Choudhary, R. Simha, B. Narahari, and N. Memon, “SAFE-OPS: An approach to embedded software security,” *ACM Trans.Embed.Comput.Syst.*, vol. 4, no. 1, pp. 189–210, 2005.

[47] O. Gelbart, P. Ott, and B. Narahari, “CODESSEAL: Compiler/FPGA approach to secure applications,” *IEEE Int. Conf. Intell. Secur. Informatics*, pp. 530–535, 2005.

[48] R. V. Steiner and E. Lupu, “Attestation in Wireless Sensor Networks : A Survey,” *ACM Comput. Surv.*, vol. 49, no. 3, pp. 1–31, 2016.

[49] D. Spinellis, “Reflection as a mechanism for software integrity verification,” *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 1, pp. 51–62, 2000.

[50] D. Schellekens, B. Wyseur, and B. Preneel, “Remote attestation on legacy operating systems with trusted platform modules,” *Sci. Comput. Program.*, vol. 74, no. 1–2, pp. 13–22, 2008.

[51] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, “Architectural support for copy and tamper resistant software,” *Asplos’00*, vol. 34, no. 5, pp. 168–177, 2000.

[52] D. Perito, G. Tsudik, and K. El Defrawy, “SMART : Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust,” *Security*, 2012.

[53] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, “TrustLite,” *Proc. Ninth Eur. Conf. Comput. Syst. - EuroSys ’14*, pp. 1–14, 2014.

[54] C. Basile, S. Di Carlo, and A. Scionti, “MobHat code mobility and reconfigurable computing joining forces for software protection.” IEEE Transactions on Systems, Man, and Cybernetics, 2009.

[55] C. Basile, S. Di Carlo, and A. Scionti, “FPGA based remote code integrity verification of programs in distributed embedded systems,” p. 13, 2011.

[56] C. Lu, T. Zhang, W. Shi, and H. H. S. Lee, “M-TREE: A high efficiency security architecture for protecting integrity and privacy of software,” *J. Parallel Distrib. Comput.*, vol. 66, no. 9, pp. 1116–1128, 2006.

[57] C. Turner, “Driving with ARMv8-R Architecture from Mobile to Automobile October 2013,” no. Ivi, pp. 1–10, 2013.

[58] Rocketboards, “Arrow SoCKit Evaluation Board,” 2016. [Online]. Available: https://rocketboards.org/foswiki/view/Documentation/ArrowSoCKitEvaluationBoard. [Accessed: 16-Dec-2016].

[59] Rocketboards, “GSRD 14.1 User Manual,” 2015. [Online]. Available: https://rocketboards.org/foswiki/view/Documentation/GSRD141. [Accessed: 25-Aug-2016].

[60] ARMKeil, “Whetstones.” [Online]. Available: http://www.keil.com/benchmarks/whetstone.asp. [Accessed: 22-Oct-2016].

[61] A. T. Services, “How to Run Industry-Standard Benchmarks on the Cyclone V SoC and Arria V SoC FPGAs,” pp. 1–14.

[62] S. Checkoway *et al.*, “Comprehensive Experimental Analyses of Automotive Attack Surfaces,” *System*, pp. 6–6, 2011.

[63] A. A. A. Al-Wosabi, Z. Shukur, and M. A. Ibrahim, “Framework for software tampering detection in embedded systems,” *Proc. - 5th Int. Conf. Electr. Eng. Informatics Bridg. Knowl. between Acad. Ind. Community, ICEEI 2015*, pp. 259–264, 2015.

[64] K. Stouffer, J. Falco, and K. Scarfone, “Guide to Industrial Control Systems (ICS) Security,” *Recomm. Natl. Inst. Stand. Technol.*, no. SP 800-82, pp. 1–157, 2007.

[65] Microsoft, “Ten Immutable Laws of Security.” [Online]. Available: https://technet.microsoft.com/en-us/library/hh278941.aspx. [Accessed: 24-Aug-2016].

[66] M. Beaumont, B. Hopkins, and T. Newby, “Hardware trojans-prevention, detection, countermeasures (a literature review),” 2011.

[67] S. Skorobogatov and C. Woods, “Breakthrough silicon scanning discovers backdoor in military chip,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 7428 LNCS, no. September, pp. 23–40, 2012.

[68] A. Baumgarten, M. Steffen, M. Clausman, and J. Zambreno, “A case study in hardware Trojan design and implementation,” *Int. J. Inf. Secur.*, vol. 10, no. 1, pp. 1–14, 2011.

[69] P. Reber and S. Graham, “Security By Design In System-On-A-Chip Applications,” in *To Appear In Proceedings of the 12th International Conference On Cyber Warfare and Security*, 2017.

[70] Altera Corporation, “Bare-Metal, RTOS, or Linux? Optimize Real-Time Performance

with Altera SoCs," no. December, p. 12, 2014.

[71] S. Designer, "Getting around non-executable stack (and fix)," *Bugtraq*, 1997. [Online]. Available: http://seclists.org/bugtraq/1997/Aug/63. [Accessed: 25-Oct-2016].

[72] Y. Choi, J. Kang, and D. Nyang, "Proactive Code Verification Protocol in Wireless," pp. 1085–1096, 2007.

[73] Y. Yang, X. Wang, S. Zhu, and G. Cao, "Distributed software-based attestation for node compromise detection in sensor networks," *Proc. IEEE Symp. Reliab. Distrib. Syst.*, pp. 219–228, 2007.

[74] A. Seshadri, M. Luk, A. Perrig, L. Van Doorn, and P. Khosla, "SCUBA : Secure Code Update By Attestation in Sensor Networks."

[75] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, "Refutation of 'On the difficulty of software-based attestation of embedded devices,'" *CCS '09 Proc. 16th ACM Conf. Comput. Commun. Secur.*, pp. 400–409, 2009.

[76] C. J. Morford, "BitMaT - Bitstream Manipulation Tool for Xilinx FPGAs," p. 60, 2005.

[77] J. Note, "From the bitstream to the netlist," *16th Int. ACM/SIGDA Symp. F. Program. gate arrays*, 2008.

[78] F. Benz, A. Seffrin, and S. A. Huss, "Bil: A tool-chain for bitstream reverse-engineering," *Proc. - 22nd Int. Conf. F. Program. Log. Appl. FPL 2012*, no. April, pp. 735–738, 2012.

[79] S. Singh and P. James-Roxby, "Lava and JBits: From HDL to Bitstream in Seconds," *9th Annu. IEEE Symp. Field-Programmable Cust. Comput. Mach.*, 2001.

[80] Altera, "Avalon Interface Specifications," p. 52, 2013.

[81] Altera, *SoCkit User Manual Rev C.* .

[82] S. Lucks, "Tree Hashing," pp. 1–29, 2013.

[83] A. Rose and B. Ramsey, "Picking Bluetooth Low Energy Locks From A Quarter Mile Away," in *DefCon*, 2016.

[84] C. Snyder, "Hackers showed us how to break into the power grid — and it was shockingly easy," 2016. [Online]. Available: http://www.businessinsider.com/redteam-hackers-power-grid-company-2016-4. [Accessed: 06-Dec-2016].

[85] E. Öksüzo\uglu and D. S. Wallach, "VoteBox nano: a smaller, stronger FPGA-based voting machine," *Proc. 2009 Conf. Electron. voting Technol. Trust. elections*, p. 8, 2009.

[86] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Des. Test Comput.*, vol. 27, no. 1, pp. 10–25, 2010.

[87] R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor, "Trustworthy hardware: Identifying and classifying hardware trojans," *Computer (Long. Beach. Calif).*, vol. 43, no. 10, pp. 39–46, 2010.

[88] B. Stone and S. Stone, "Comparison of Radio Frequency Based Techniques for Device Discriminiation and Operation Identification," in *11th International Conference On Cyber Warfare And Security*, 2016.

[89] E. Skoudis and J. Ullrich, “The Seven Most Dangerous New Attack Techniques and What â€™s Coming Next SANS Curriculum Lead for Founder of Counter Hack.”

[90] C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang, “ReDaS: Remote attestation to dynamic system properties: Towards providing complete system integrity evidence,” *Proc. Int. Conf. Dependable Syst. Networks*, pp. 115–124, 2009.

[91] T. Abera *et al.*, “C-FLAT: Control-FLow ATtestation for Embedded Systems Software,” *Proc. 2016 ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016.

[92] C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang, “Remote attestation to dynamic system properties: Towards providing complete system integrity evidence,” *Proc. Int. Conf. Dependable Syst. Networks*, pp. 115–124, 2009.

[93] D. Zhang and D. Liu, “DataGuard: Dynamic data attestation in wireless sensor networks,” *Proc. Int. Conf. Dependable Syst. Networks*, pp. 261–270, 2010.

[94] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, “Control-flow integrity principles, implementations, and applications,” *Tissec*, vol. 13, no. 1, pp. 1–40, 2009.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From – To)*
03-23-2017	Master's Thesis	August 2015 – March 2017

TITLE AND SUBTITLE

A Comprehensive Security Analysis of and an Implementation Framework For Embedded Software Attestation Methods Leveraging FPGA-Based System-On-A-Chip Architectures

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Reber, Patrick A., 2d Lt, USAF

5d. PROJECT NUMBER
17G385

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way, Building 640
WPAFB OH 45433-8865

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-17-M-063

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory
2241 Avionics Circle WPAFB OH 45433
Attn: Lt Col Patrick Sweeney
Patrick.Sweeney@us.af.mil
937-713-4252

10. SPONSOR/MONITOR'S ACRONYM(S)

AFRL/RYWA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRUBTION STATEMENT A. APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES
This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

As embedded devices continue to proliferate, the safety and security of critical applications increasingly relies on trust in the running firmware. Malicious actors who compromise these devices may alter this code to accomplish their objectives while remaining undetected. Integrity checks of this code alerts users to changes which indicate a potential attack. This is commonly referred to as software attestation, and utilizes a challenge-response protocol that allows a trusted verifier to integrity check the memory of an untrusted device.

This research assesses the potential for utilizing FPGA System-on-a-Chip (SoC) architectures for software attestation and analyzes the resulting security. A framework is proposed to leverage the SoC capabilities to quickly and easily perform software attestation with minimal runtime impact. The SoC hardware acts as a trusted local entity which aids in verification of the firmware run by the processor, while the incorporation of the SoC requires little processor software or configuration changes. This allows designs which previously utilized solely microcontrollers to easily port to this architecture for increased software security.

This framework is constructed with an example algorithm and confirmed to detect attacks. The resulting degradation in processor speed is examined, as well as the potential attestation speed. Processor degradation ranges from 0.5 to 42 percent and read speed ranges from 0.06 to 3.2GB/s. Multiple additional alternatives are implemented and analyzed, which increase security at the cost of speed and simplicity. Various potential options are characterized in a novel attestation taxonomy extension for comparison.

15. SUBJECT TERMS
Software Attestation, Embedded Software Security, FPGA, System-On-A-Chip

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Scott Graham, AFIT/ENG
U	U	U	UU	121	19b. TELEPHONE NUMBER *(Include area code)* (937) 255-3636 (Scott.Graham@afit.edu)

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18