SSL Hardware Hiding:

Increasing the Security of OpenSSL Through Tightly-Coupled FPGA Hardware

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Masters of Science

by

BRETT NICHOLAS

Thayer School of Engineering

Dartmouth College

Hanover, New Hampshire

November 2017

Examining Committee:

Chairman_____
Stephen Taylor, Ph.D.

Member_____
Eric Hansen, Ph.D.

Member_____
Eugene Santos, Ph.D.

_____
F. Jon Kull
Dean of Graduate Studies

## Abstract

This paper explores a novel approach to improving the security and performance of OpenSSL by hiding encryption algorithms and keys within the perimeter of a single System-on-Chip (SoC) device. The approach only recently became feasible with the introduction of a generic hardware acceleration API into the OpenSSL software suite. Although the API is intended to facilitate the use of generic encryption co-processors, we instead use it to establish a *single-chip, hardware base-of-trust* that exposes no off-chip interconnects to reverse-engineering. To achieve this base-of-trust, we hide popular encryption algorithms and associated keys within on-chip Field-Programmable Gate Array (FPGA) technology. Consequently, compromised applications running on the SoC embedded processors are unable to inspect the details of encryption operations or gain access to keys. Performance enhancements are viable through high-performance on-chip bus interconnects that couple the FPGA fabric directly into the memory hierarchy of the associated processors. To demonstrate the approach, we hide custom SHA, AES, and RSA algorithms within the FPGA fabric of a Xilinx Zynq SoC. Performance enhancement is quantified through OpenSSL's built in performance monitoring tools, in conjunction with a series of test programs using the OpenSSL EVP high-level cryptography API. Using only un-optimized, proof-of-concept hardware algorithms, the proposed system replicates the functionality of the OpenSSL library, while exposing no algorithms or keys to the software running on the processors. Core algorithm performance was shown to be up to 40x faster in hardware than the equivalent software-only stack, but with a net performance penalty on a systems level due to memory access bandwidth and context switching.

## Acknowledgements

## Table of Contents

## List of Figures

## *List of Acronyms*

| | |
|---|---|
| Advanced Microcontroller Bus Architecture | AMBA |
| Application Programmer Interface | API |
| Application Processor Unit | APU |
| Advanced eXtensible Interface | AXI |
| Block Random Access Memory | BRAM |
| Board Support Package | BSP |
| Computer Automotive Network | CAN |
| Cipher Block Chaining | CBC |
| Configurable Logic Block | CLB |
| Command Line Interface | CLI |
| Count Lines Of Code | CLOC |
| Carry-Propagate Adder | CPA |
| Central Processing Unit | CPU |
| Carry-Save Adder | CSA |
| Double Data Rate (memory) | DDR |
| Direct Memory Access | DMA |
| Dynamic Random Access Memory | DRAM |
| Digital Signal Processing | DSP |
| Electronic CodeBook | ECB |
| Electrically Erasable Programmable Read Only Memory | EEPROM |
| Envelope (OpenSSL API interface) | EVP |
| Federal Information Processing Standards | FIPS |
| Field Programmable Gate Array | FPGA |
| GNU's Not UNIX (free software collection) | GNU |
| General Purpose Input/Output | GPIO |
| Hardware Description Language | HDL |

| | |
|---|---|
| High Level Synthesis | HLS |
| Input/Output | IO |
| Input/Output ConTroL | IOCTL |
| Input/Output Peripherals | IOP |
| Internet Protocol | IP |
| Intellectual Property | IP |
| Loadable Kernel Module | LKM |
| Media Access Controller | MAC |
| OpenEmbedded | OE |
| Open Systems Interconnect | OSI |
| Public Key Cryptography Standards | PKCS |
| Programmable Logic | PL |
| Processing System | PS |
| Random Access Memory | RAM |
| Rivest, Shamir, Adleman (cryptographic algorithm) | RSA |
| Register Transfer Logic | RTL |
| Secure Hash Algorithm | SHA |
| Source Lines of Code | SLOC |
| Serial Peripheral Interface | SPI |
| Static Random Access Memory | SRAM |
| Secure SHell | SSH |
| Secure Sockets Layer | SSL |
| Transportation Control Protocol | TCP |
| Transportation Layer Security | TLS |
| Universal Asynchronous Receiver/Transmitter | UART |
| Universal Serial Bus | USB |
| Virtual Private Network | VPN |

## Chapter 1: Background and Motivation

**Problem:** How can the performance and security of TLS be improved?

**Hypothesis:** Both Performance and Security of TLS can be improved by leveraging Commodity System-on-Chip (SoC) Architectures.

### Transportation Layer Security (TLS)

TLS is one of the most commonly used cryptographic protocols that provides secure communication over the Internet. Often referred to by the name of its predecessor, Secure Sockets Layer (SSL), TLS allows client/server applications to communicate in a manner that prevents eavesdropping, tampering, or message forgery (1). The most popular implementation of TLS is an open source library – OpenSSL – used by over two thirds of all web servers on the internet (2). OpenSSL is also the default TLS library shipped on most Linux distributions.

Although the TLS protocol specification itself is considered highly secure, significant vulnerabilities in its implementation, such as the 2014 "Heartbleed" bug (3), have clearly demonstrated that secure protocol specifications do not guarantee security if they are implemented with vulnerable software. Heartbleed allowed anyone on the internet to read the memory of an affected OpenSSL client system, thereby compromising the secret keys used to identify service providers and encrypt traffic. This exposure had the effect of releasing the names and passwords of client computers, and exposing private traffic. Even if OpenSSL itself is not exploited, other software present on a host computer can easily negate the privacy benefits of OpenSSL by providing a privilege escalation path that

enables an adversary to obtain super-user, or *root*, access to a host (4). This is of particular concern in networking and connectivity applications, since compromise of a single client may allow malicious access to the entire network.

Recently, the availability of special purpose encryption/decryption co-processors has prompted an evolution in the OpenSSL implementation: The high-level cryptographic Application Programming Interface (API), known within OpenSSL as the envelope (EVP) interface, has been re-organized to include a new API called the "Engine API" that supports alternative cryptography implementations and hardware acceleration. Third party vendors can request that their engines be included in the OpenSSL source tree, or the Engine code can be loaded dynamically at runtime as a shared library through a special built-in engine called "dynamic".

For example, the EVP functions for using a generic encryption cipher are shown below:

```
 1 int EVP_EncryptInit_ex(EVP_CIPHER_CTX *ctx,const EVP_CIPHER *cipher,
 2                         ENGINE *impl,
 3                         const unsigned char *key,
 4                         const unsigned char *iv);
 5
 6 int EVP_EncryptUpdate( EVP_CIPHER_CTX *ctx, unsigned char *out,
 7                        int *outl, const unsigned char *in, int inl);
 8
 9 int EVP_EncryptFinal(EVP_CIPHER_CTX *ctx,unsigned char *out,
10                       int *outl);
```

All low-level cipher operations must be invoked through these three functions, regardless of the chosen cipher (AES-256, Blowfish, 3DES, etc.). A specific cipher is selected by passing a pointer to the cipher's associated data structure as an argument to the `EVP_EncryptInit_ex()` function. Since the introduction of the engine API, a pointer to an engine object can also be passed as an argument to the `EVP_EncryptInit_ex()`

function, causing the EVP interface to choose the engine's cipher implementation instead of the default OpenSSL cipher. A similar interface is available to all the core cryptographic operations, in particular for SHA-256 digests, symmetric key cryptography with AES-256, and public key cryptography with RSA-2048. An engine implementation can also be invoked from the OpenSSL command line interface by adding an additional command line flag to the desired operation, and providing a path to the engine shared library. The following example shows how to use an engine's implementation of the SHA-256 digest algorithm to compute the digest of the string "hello!" using the standard OpenSSL CLI:

```
echo "hello!" | openssl dgst -engine /path/to/libengine.so -sha256
```

More specific details of the engine API will be discussed in Chapter 4.

**Field Programmable Gate Arrays**

Field Programmable Gate Arrays (FPGAs) are semiconductor integrated circuits (ICs) that can be reprogrammed to change or augment the functionality of the implemented circuit after manufacturing; this distinguishes them from Application Specific Integrated Circuits (ASICs), which are custom manufactured for each design (5). FPGA logic consists of arrays of configurable logic cells, interspersed with resources such as block RAMs, multipliers, and high-speed communications blocks (Figure 1). The resources are connected by a rich fabric of programmable interconnects. FPGAs are configured, or "programmed" by populating lookup tables, block RAMs, and other aspects of the logic resources possessing "state", as well as by specifying the routing of signals between the logic resources.

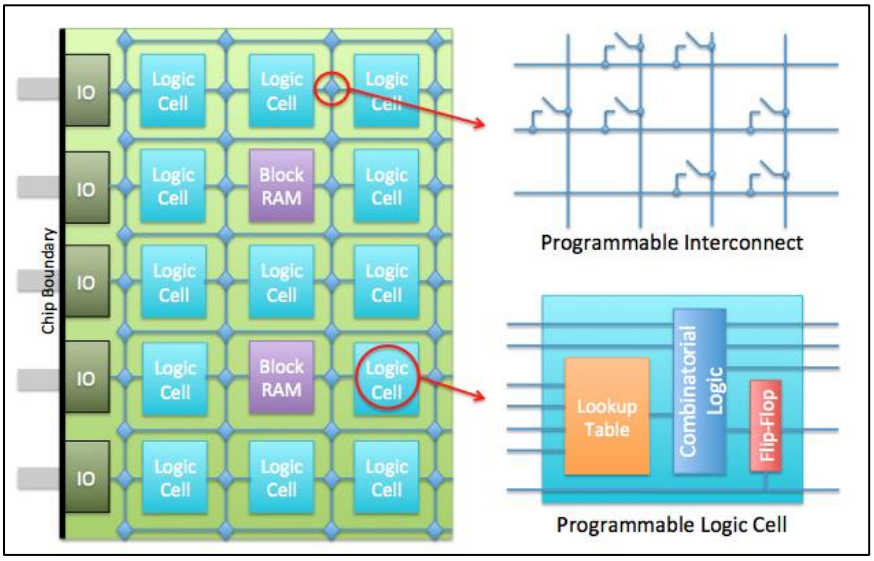


**Figure 1:** Field Programmable Gate Array Architecture

FPGA logic configuration is generally described using a hardware description language (HDL), such as VHDL or Verilog. HDLs are transformed via synthesis and implementation tools into a *bitfile* for use in configuring the FPGA logic. This process is analogous to software program descriptions in C language modules that are compiled and linked into executable images. The bitfile is typically loaded using a trusted boot process on system start-up, although in some architectures the FPGA logic can be partially reconfigured during system run-time. In all cases FPGA users gain software-like development and deployment flexibility while maintaining the performance and deterministic behavioral benefits of hardware-based design.

Only recently, however, has FPGA logic been co-located with high-performance processors within the physical chip boundary. Prior to this, FPGA logic has been of little utility in processor security applications due to the exposed communications buses and limited coupling between discrete processor and FPGA chips. On-chip colocation,

combined with tight coupling between processing cores and the FPGA logic, creates the opportunity for a new class of techniques for system security based on the ability to *hide critical control logic and data structures from the processing cores using tightly-coupled FPGA logic*.

**Heterogeneous System-on-Chip Architectures**

Xilinx Inc., the largest FPGA manufacturer by market share (6), released the first commercially available silicon devices co-locating high-performance, multi-core processors with integrated on-chip FPGA logic in 2012. Subsequently, following its purchase of Altera, Intel has announced its intension to pursue similar offerings (7). Unlike prior FPGAs with supporting on-chip processors intended to offload processing tasks from the FPGA (such as a hosting a TCP/IP protocol stack), these new System-on-Chip (SoC) devices were "processor-centric" systems. These architectures were designed with *hardware acceleration of software algorithms* in mind.

Hardware acceleration is a technique for speeding up algorithms by implementing directly them in FPGA logic. Many digital signal processing algorithms, for example, enjoy significant (typically 10-50-fold) performance speedups when implemented in FPGA logic. Prior to 2012 system designers typically had to integrate two chips on a circuit board: a processor and, separately, an FPGA. They then had to provide a high performance *external* data bus, vulnerable to monitoring and reverse-engineering, between the two to provide data sharing. The overhead associated with transmitting data across the external bus between the processor and the FPGA is, from a design perspective, usually the limiting factor in the utility of this technique. The development costs associated with high

performance bus design, and its associated drivers, as well as the manufacturing costs associated with populating two high pin-count/high pin-density chips, were often limiting factors in the design process.

The internal architectures of these new generation of chips are designed to surmount this performance bottleneck, achieved by leveraging a variety of bus-interconnects that allow high-performance data transfers between the FPGA and processors. Multiple interconnect options address bandwidth and latency considerations associated with a given application's data transfer requirements and the data's location in the system. Sharing the same die reduces both the component and assembly costs associated with the discrete equivalent. This opens many new opportunities for hardware acceleration that were previously impractical due to interconnect limitations or cost concerns.

Figure 2 provides a high-level block diagram of Xilinx's Zynq System-on-Chip (SoC) architecture used in this thesis. The Zynq integrates ARM's Cortex-A9MP dual-core processor architecture with rich FPGA logic resources. The dual-core processors include hardware floating-point support and clock rates up to 800 MHz. The processors have independent 32kbyte instruction and data level-1 caches, and they share a 512k level-2 cache, supporting up to 2GB of off-chip DDR memory, with a 256kbyte on-chip static RAM. On-chip peripherals, dedicated to the processors, include dual gigabit Ethernet, serial communications, and direct memory access (DMA), including channels dedicated to processor-FPGA transfers, and timer resources. A hardware-accelerated cryptographic engine and analog to digital converter are also available to both the processors and the

FPGA logic. The Zynq product family's FPGA logic resources scale from 28k logic cells (ZC7010) to 444k logic cells (ZC7100), with dedicated block RAMS ranging from 240KB to 3MB and between 80 and 2020 DSP slices. **One particularly important aspect of this architecture, is that the FPGA may act as a "bus-master", meaning that it controls the processors ability to access data stored in the FPGA, effectively hiding data within the FPGA such that it is not visible, *even with root access on the processor.***

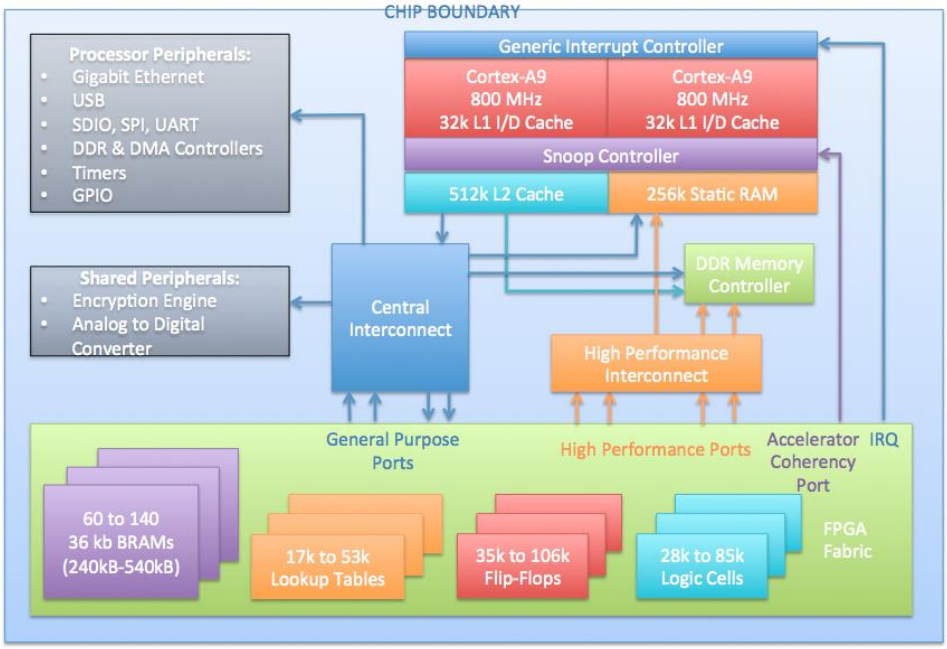


**Figure 2:** Xilinx Zynq Simplified Block Diagram

### Hardware Synthesis

One of the core impediments to employing FPGA hardware acceleration has been the complexity and cost of hardware design, which is typically achieved with Hardware Description Languages (HDLs) such as Verilog and VHDL, and toolchains such as the Xilinx Vivado suite. The introduction of the Zynq, however, has heralded a major evolution of design tools employing High-Level Synthesis (HLS): The Vivado tool-suite, provided by Xilinx, automatically generates synthesizable HDL code from system code written in

the C programming language. Following the transition from assembly code to high-level languages in software design, this tool-suite opens the door to decreased maintenance costs and increased flexibility for hardware designers. **In the context of this thesis, the goal is to employ HLS to ensure that all the core cryptographic blocks can be re-used or implemented concurrently in hardware for use by different software processes**.

**Approach**

Recall, from the problem and hypothesis definitions, that this thesis explores the use of commodity SoC devices to increase the security of OpenSSL. The core approach is to hide encryption algorithms, hashes, and keys within the perimeter of a single System-on-Chip (SoC) device, such as the Zynq device.

The approach became feasible with the introduction of the generic engine API into OpenSSL, introduced earlier in this section; Although the API was intended to facilitate the use of generic encryption co-processors, here it is used to establish a *single-chip, hardware base-of-trust* that exposes no off-chip interconnects to monitoring and reverse-engineering. To achieve this base-of-trust, popular encryption algorithms, and their associated hashes and keys, are hidden within on-chip FPGA logic. Consequently, compromised software, running on the SoC embedded processors, are unable to inspect the details of encryption operations or gain access to keys, even if they are able, through privilege escalation, to achieve root-level privileges on a processor. Performance and security enhancements are viable through the high-performance on-chip bus interconnects that couple the FPGA fabric directly into the memory hierarchy of the associated processors. To demonstrate the approach, custom SHA, AES, and RSA algorithms are

embedded into the FPGA fabric of a Xilinx Zynq SoC running a minimal Linux distribution. The implementation uses the hardware acceleration API along with custom driver software to integrate the hardware algorithms into OpenSSL. All software runs on a customized minimal Linux distribution, which supports the network stack and memory management required by OpenSSL. Linux was chosen due to its widespread use, compatibility with OpenSSL, and ease of customization. All hardware design is conducted in HLS; consequently, the performance results presented in this report represent a lower-bound which can only be improved by replicating HLS functional blocks for various applications.

## Metrics

### *Security*

Unfortunately, it is difficult to quantify the discrete improvement in security provided by the approach, especially since the expected enhancements purport to protect against zero-day attacks, which are, by their very nature, unknown. Consequently, this thesis instead quantifies the proportion of the SSL library, in source-lines-of-code (LOC), that is hidden in hardware, rendering it inaccessible to modification and compromise by malicious software executing on the processors. This methodology is informed by recent studies showing that the number of vulnerabilities in open-source software is directly proportional to the number of lines of source code (8), with an average of 0.61 programming defects found for every thousand lines of open-source software (9). Rather than providing an absolute measure of effectiveness, this measure quantifies the expected level of improvement or hardening, contributing to improved resilience. Moving core cryptographic operations from addressable RAM into FPGA fabric decreases visibility

from application and kernel software; even if attacker code is elevated to the highest processor privilege level it can only observe what the FPGA logic designer exposes on the processor's memory bus, and the processing system cannot modify the FPGA logic's configuration unless explicitly allowed to do so (10).

*Performance*

Performance enhancement due to hardware acceleration in the FPGA is assessed using the standard metrics of latency and throughput. Latency is defined as the time required for an input to the system to provide a valid result, while throughput is defined as the number of such valid input/output actions that can be processed per unit time (11). These metrics are obtained for each hardware block at both the "block-level" and the "system-level". Block-level performance reflects the functionality of each hardware implementation as tested in isolation. Results are obtained using a custom unit test harnesses, and the post-synthesis performance analysis tools in the Vivado design suite. System-level metrics demonstrate the overarching performance of the entire system once it is fully integrated with OpenSSL. These metrics are acquired using OpenSSL's built-in performance benchmarking tools to compare cryptographic functionality against each algorithm's default software implementation and to reveal any overhead associated with the engine API, and hardware data transfer costs.

*Logic Utilization*

FPGA logic resource utilization is presented as a relative measurement of implementation cost and gives measure for the number of replicated functional blocks it would be possible to use for performance enhancement. These metrics are treated as relative measurements for two reasons: 1) hardware implementations can be optimized based on designer

specified constraints, including resource utilization, which will affect findings, and 2) the results reported are for an entry-level part (ZC7020 with 85k logic cells) in the processor family. 50% FPGA resource utilization in this part when optimized for performance might only consume 10% FPGA resource utilization in the larger ZC7100 (with 444k logic cells). However, 30% utilization for design A vs. 10% utilization for design B can be used as a relative measurement of 3x cost for design A over design B.

**Roadmap**

The remainder of this thesis is divided into the following chapters:

*Ch. 2: High level system architecture* provides a systems-level description of the design proposed in this thesis, and details how each component is interconnected.

*Ch. 3: Hardware Design and Algorithm Synthesis* details the design and implementation of each cryptographic core in FPGA hardware.

*Ch. 4: Operating System and OpenSSL Integration* specifies how the cryptographic hardware presented in Ch. 3 interfaces with the Linux operating system and the OpenSSL software stack.

*Ch. 5: Testing and Performance Analysis* quantifies the performance and efficacy of the proposed design through unit testing and the use of standardized performance analysis tools.

*Ch. 6: Future Work and Conclusions* makes concluding remarks on the work presented in earlier chapters and describes directions for future work.

# Chapter 2: High Level System Architecture

Recall that the goal of this thesis is to improve the security of OpenSSL by hiding encryption algorithms and keys within the perimeter of a single System-on-Chip (SoC) device, establishing a *single-chip, hardware base of trust* that is unable to be exploited via reverse engineering of any off-chip interconnects. This chapter will introduce the key architectural features of the Xilinx Zynq SoC and provide a high-level overview of the proposed design, paying specific attention to the hierarchical structure of the design and the interconnectedness of each component.

## Xilinx Zynq Architectural Description

The Zynq SoC is divided into two distinct subsystems: The Processing System (PS), and the Programmable Logic (PL). The two subsystems are connected by the Advanced Microcontroller Bus Architecture (AMBA) interconnect, which resides in the PS. Figure 3 shows a high-level block diagram of the Zynq SoC architecture, with the PS colored light green and the PL in yellow.

### *Programmable Logic (PL)*

The Zynq PL contains FPGA fabric similar to the Xilinx 7-series Artix FPGAs, with the addition of several special purpose ports and buses that tightly couple it to the PS. It consists of an interconnected matrix of Configurable Logic Blocks (CLBs), Block RAM (BRAM), and Digital Signal Processing (DSP) slices. CLBs are the main logic resources for implementing sequential and combinatorial logic circuits. (12). Each CLB element comprises eight six-input look-up tables (LUTs), 16 flip-flops, 256 bits of distributed

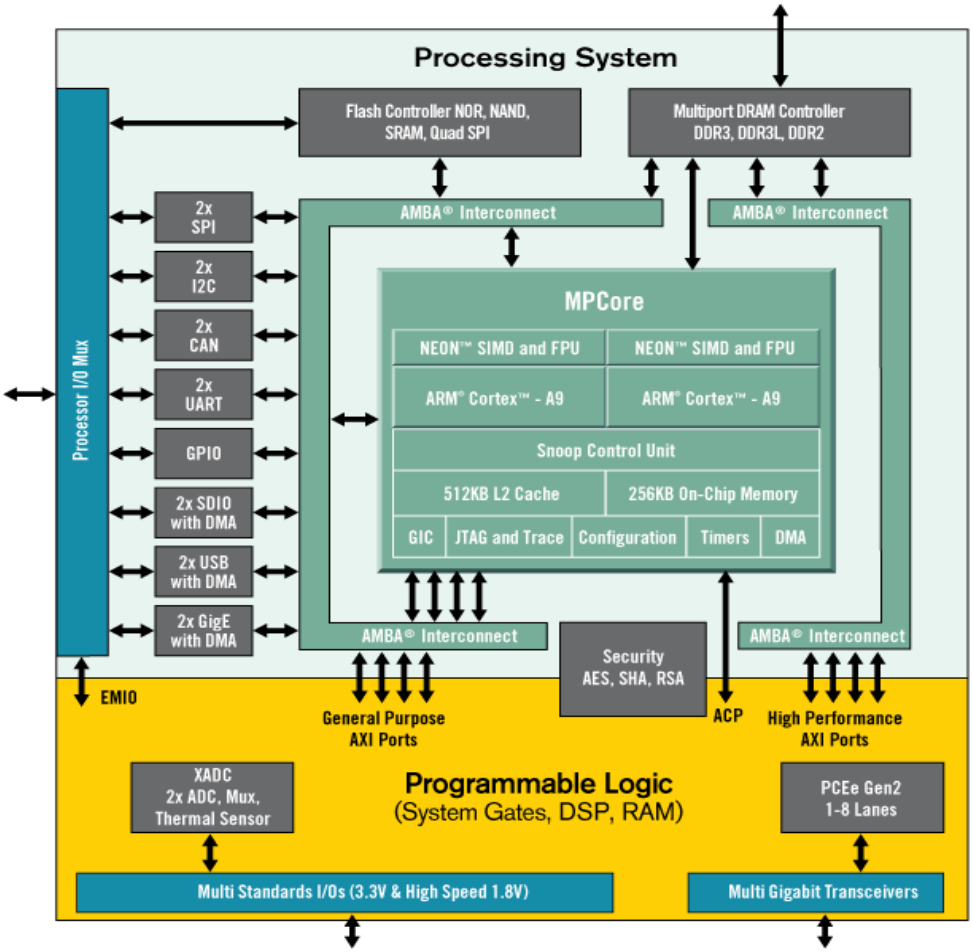


**Figure 3:** Zynq-7000 Architecture Block Diagram

RAM, and 128 bits of DRAM-based shift registers, and is connected to a switch matrix that provides access to the FPGA's global routing resources. The Zynq PL also contains 140 35Kb blocks of dual-port BRAM, which serve as the primary storage elements, and 220 DSP slices, which facilitate low-latency parallelized implementations of computationally intensive operations like binary multiplication and multiply-accumulate.

*Processing System (PS)*

The PS is the Zynq subsystem that contains the ARM processors, associated peripherals, and memory resources. It consists of four major functional blocks: the application

processor unit (APU), the ARM AMBA AXI Interconnect, the memory interfaces, and the I/O peripherals (IOP).

The APU houses the dual-core ARM Cortex-A9 processors, associated L2 cache memory, the accelerator coherency port interface enabling cache-coherent accesses from PL to CPU memory space, 256K of on-chip RAM, a generic interrupt controller, and several CPU timers.

The Memory interfaces are hardware controllers for the various types of flash memory and DRAM that the Zynq is compatible with. They are connected to the processors and PL as slaves addressable through the AMBA interconnect, which is detailed later in this section.

The Zynq IOP are a subset of the standard set of peripheral communication controllers that can be found on most modern microcontrollers, including SPI, I2C, CAN, UART, GPIO, SDIO, USB, and Gigabit Ethernet. The peripheral controllers are connected to the processors as slaves via the AMBA interconnect, and contain readable/writable control registers that are addressable in the processors' memory space (13).

The APU, memory interface unit, IOP, and PL are all connected to each other through a multilayered bus infrastructure called the ARM Advanced Microcontroller Bus Architecture Advanced eXtensible Interface 4 (AMBA-AXI4) interconnect. The "*AMBA"* label refers to the physical architecture of the interconnect, however the communications bus itself is usually referred to by the protocol specification that it uses: the AXI 4 protocol.

This paper uses the term "*AMBA interconnect"* and "*AXI bus"* interchangeably. AXI4 is non-blocking and supports multiple simultaneous master-slave transactions. The bus is designed such that latency sensitive masters, such as the ARM CPU, have the shortest paths to memory, and bandwidth critical masters, such as the potential PL masters, have high throughput connections to the slaves with which they need to communicate (14). In most instances the FPGA logic is intended to act as the bus master – initiating read and write transactions on the bus. This design approach is critical to the notion of "hiding in hardware". Only data presented to the processing system in the form of an AXI4-bus interface is visible to the processing system at any level of privilege, and only data exposed over a subset of the general-purpose ports can be requested by the processor – all other data

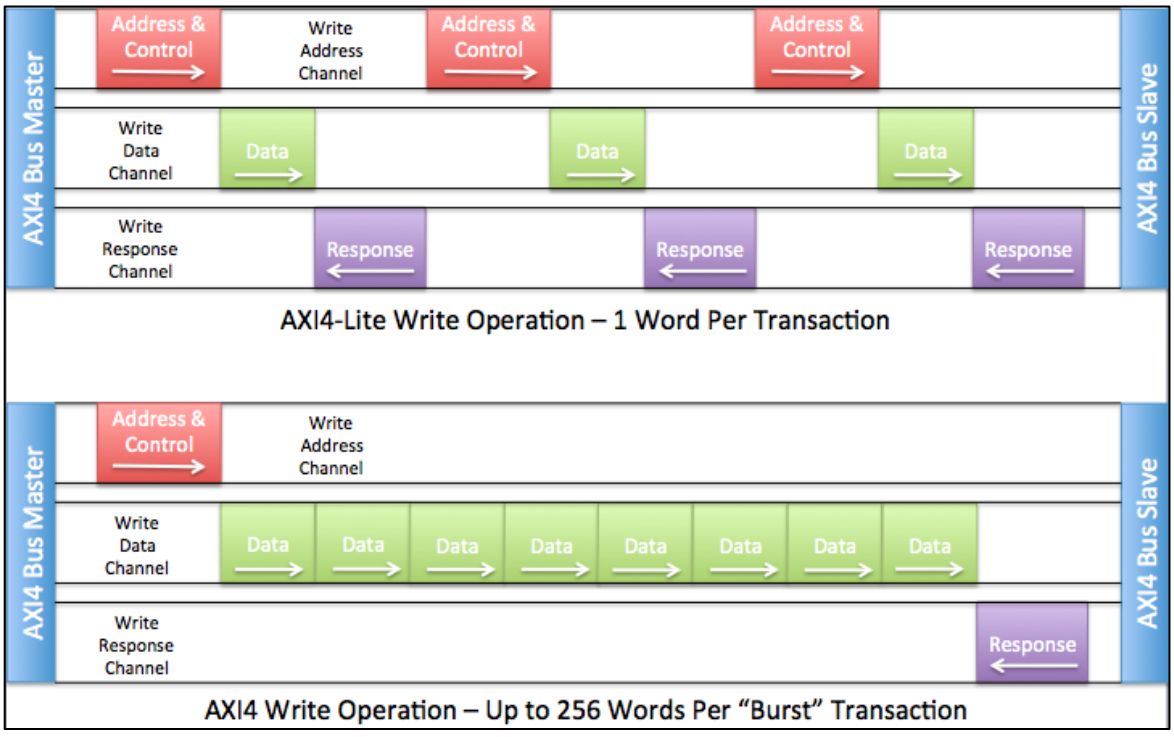


**Figure 4: AXI4-Lite and AXI4 Write Operations**

transfers between the FPGA and the processing system are driven by control logic in the FPGA (15).

The AXI ports support two types of interfaces: AXI4-lite and full AXI4 (Figure 4). The two interface types are differentiated by their support for burst transactions. All AXI4 read transactions require two phases: the control phase, where read address information is sent, and the data phase, where the requested data is returned. AXI4 write transactions require three phases: control, data, and response. Write address information is sent during the control phase, the data to write is sent during the data phase, and transaction status (success or error) is returned in the response phase. Both interfaces support burst transactions, which allow for up to 256 data words to be transferred to/from sequential memory addresses in shared memory space. The AXI4-lite implementation requires two cycles per word read (one for control, one for data), or 50% transaction overhead. The full AXI4 implementation can read 256 sequential words in 256+1 cycles, less than 0.5% overhead. Therefore, on a 64-bit wide bus operating at the upper limit of the FPGA logic's clock capabilities (200 MHz), full AXI4 and AXI4-lite bursts would yield a throughput of approximately 12 gigabits per second (Gbps) and 6 Gbps, respectively (15). For pure data transfer performance, burst memory reads are clearly the optimal solution. However, the FPGA side of the bus interface is instantiated in FPGA logic resources, so there is a higher resource utilization cost associated with implementing the more sophisticated bus interface. Ultimately, the specific requirements of a design will dictate which is the optimal implementation.

It should be noted that the system developed in this thesis uses hardware cores instantiated as AXI slaves, rather than AXI masters, due to the simplicity of implementation and integration into the rest of the software stack. Most notably, having the hardware cores

function as slaves greatly simplified integration of the hardware into OpenSSL, and was deemed an acceptable tradeoff for a first attempt at a proof-of-concept system, despite the performance penalty incurred. As is detailed in chapters 5 and 6, converting the hardware algorithms from AXI slaves to AXI masters is the logical next step towards improving the proposed design, and will dramatically increase the overall performance and security of the system.

**Design Overview**

A high-level block diagram of the proposed design is shown in Figure 5. Each block with a white background is a component specifically developed as a part of this thesis. The components in the design can be organized into three distinct layers: the hardware layer, the kernel layer, and the application layer.

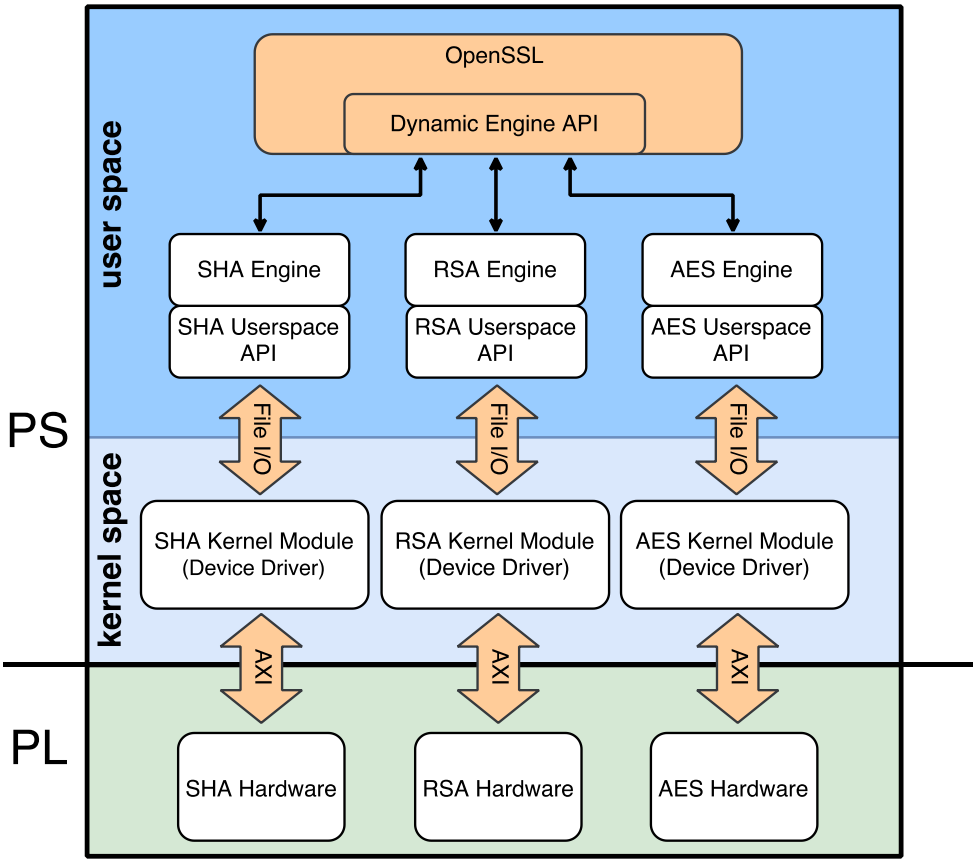


**Figure 5:** High-level Block Diagram

The hardware layer consists of the design components that reside in the PL, and is represented in Figure 5 by the green shaded region. The kernel and application layers both reside in the PS, and are represented by the light blue and dark blue shaded regions, respectively. Since RSA and AES keys, as well as SHA hashes are transitory and time-sensitive, employed purely for session keys and digests, they must be accessible from software. These vulnerabilities can easily be mitigated by moving the network interface into the FPGA alongside cryptographic functions but is beyond the scope of this thesis. The decision when to transfer a private key into hardware can be controlled by the FPGA either as a bus-master, as is proposed in a future system, or as a bus-slave configured via an out-of-band channel, such as an external network or serial device interface in the FPGA; here bit stream generation in HLS is used to effect key binding. In both cases, key transfer should be unobservable to the processor. Eventually, vulnerabilities associated with the device drivers and operating system can be removed by employing a micro-kernel and embedding it with the FPGA (15).

Although Figure 5 shows a single instance of each encryption block, reconfiguration as introduced in Chapter 1, allows multiple instances of the blocks to be employed to improve performance; for example, one complete set might of blocks can be used for encryption and another for decryption.

*Hardware Layer*

The hardware layer contains the hardware cryptographic functions synthesized directly from C code using Vivado High-Level Synthesis (HLS). Recall from chapter 1 that the functions to be implemented in hardware were the SHA256 message digest algorithm, the 256-bit version of the AES symmetric cipher, and the RSA 1024 asymmetric public key cipher. These were chosen because they are the three most basic algorithms required to establish a secure TLS session using OpenSSL. Each hardware function exists as a slave on the AXI bus, and has a bank of control and data registers that are mapped into the address space of the processors. This allows software running on either processor core to interact with the custom hardware in the same manner as any other peripheral controller on the AXI bus, such as the UART or GPIO banks. As mentioned earlier in this chapter, the hardware cores will eventually converted to act as bus-masters, able to initiate processor memory accesses without processor awareness.

*The Kernel Layer*

Recall from chapter 1 that an operating system is necessary to provide the network stack and other low-level support required by OpenSSL. A minimalist custom Linux distribution was created to serve as the target operating system, given its widespread popularity and compatibility with OpenSSL. The kernel layer comprises the loadable kernel module (LKM) device drivers that allow applications running in Linux user space to access the memory-mapped custom FPGA hardware. Device drivers for the hardware cryptographic functions must be written as LKMs because, like most operating systems, user space programs use virtual memory; they cannot access the specific physical memory addresses of a device without the kernel explicitly mapping it into the process's virtual memory. A

LKM device driver exists for each cryptographic function present in the hardware layer. Each driver serves the purpose of mapping the device registers into virtual memory. With the device drivers loaded into the kernel, user space processes can interact with the hardware through the standard Linux device file interface. The design of the kernel layer components and the operating system itself will be described in Chapter 4: Operating System and OpenSSL integration

*The Application Layer*

The application layer contains all remaining software required to integrate the hardware cryptographic blocks into OpenSSL. This consists of two additional components for each cryptographic function: a user space API, and an OpenSSL engine. The user space APIs abstract the low-level details of interacting with the LKM, providing a more logical and intuitive set of functions to control the hardware instead of using the standard Linux device file operations (open, close, read, write). The API is implemented as both a shared and static library; While not strictly necessary, this improve the modularity of the design and allow the hardware blocks to be used outside the context of OpenSSL. The OpenSSL engines are what facilitates the integration of all other components of the design into OpenSSL. Engines are compiled as a shared library, and dynamically loaded at runtime by OpenSSL when an engine is requested. Each engine must define several template functions that are required by the EVP interface to complete the respective cryptographic operations that otherwise would be performed in software. The internals of the application layer components will be further discussed in chapter 4.

# Chapter 3: Hardware Design and Algorithm Synthesis

This section describes the low-level implementation details of each hardware algorithm, and provides a systems-level overview of the hardware design. Recall from Chapter 2 that hardware layer contains the hardware cryptographic functions synthesized directly from un-optimized C code using Vivado High-Level Synthesis (HLS). The functions to be implemented in hardware are the SHA256 message digest algorithm, the 256-bit version of the AES symmetric cipher, and the RSA 1024-bit asymmetric public key cipher. These were chosen because they are the three most basic algorithms required to establish a secure TLS session using OpenSSL. Each hardware function exists as a slave on the AXI bus, and has a bank of control and data registers that are mapped into the address space of the processors.

### SHA256 Overview

SHA256 is a member of the Secure Hash Algorithm (SHA)-2 family of cryptographic hash algorithms, originally developed by the U.S. National Security Agency. A cryptographic hash algorithm (alternatively, hash "function") is a mathematical operation designed to provide a random mapping from a string of binary data to a fixed-size “message digest”. (16). The hashing operation is a one-way function, meaning that a fixed-length hash value can be generated from any piece of data, but the data cannot be reconstructed from the hash without trying every single possible input combination. This property allows a cryptographic hash algorithm to validate a message’s integrity: any change to the message will, with a very high probability, result in a different message digest (17).

**SHA256 Algorithm Detail**

The SHA256 algorithm operates on a 512-bit message block and a 256-bit intermediate hash value. It is essentially a 256-bit block cipher algorithm which encrypts the intermediate hash value using the message block as key. The algorithm uses the following basic operations (18):

- Boolean operations AND, XOR and OR, denoted by $\wedge$, $\oplus$ and $\vee$, respectively
- Bitwise complement, denoted by $\bar{}$
- Integer addition modulo $2^{32}$, denoted by A+B

Each of these operates on 32-bit words. For the last operation, binary words are interpreted as integers written in base 2.

- RotR(A, n) denotes the circular right shift of n bits of the binary word A
- ShR(A, n) denotes the right shift of n bits of the binary word A
- A||B denotes the concatenation of the binary words A and B

The algorithm uses these operations inside of the following functions:

$$Ch(X,Y,Z) = (X \wedge Y) \oplus (X \wedge Z)$$

$$Maj(X,Y,Z) = (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z)$$

$$\Sigma_0(X) = RotR(X,2) \oplus RotR(X,13) \oplus RotR(X,22)$$

$$\Sigma_1(X) = RotR(X,6) \oplus RotR(X,11) \oplus RotR(X,25)$$

$$\sigma_0(X) = RotR(X,7) \oplus RotR(X,18) \oplus ShR(X,3)$$

$$\sigma_1(X) = RotR(X,17) \oplus RotR(X,19) \oplus ShR(X,10)$$

and the 64 binary words $K_i$ given by the 32 first bits of the fractional parts of the cube roots of the first 64 prime numbers:

```
0x428a2f98 0x71374491 0xb5c0fbcf 0xe9b5dba5 0x3956c25b 0x59f111f1
0x923f82a4 0xab1c5ed5 0xd807aa98 0x12835b01 0x243185be 0x550c7dc3
0x72be5d74 0x80deb1fe 0x9bdc06a7 0xc19bf174 0xe49b69c1 0xefbe4786
0x0fc19dc6 0x240ca1cc 0x2de92c6f 0x4a7484aa 0x5cb0a9dc 0x76f988da
0x983e5152 0xa831c66d 0xb00327c8 0xbf597fc7 0xc6e00bf3 0xd5a79147
0x06ca6351 0x14292967 0x27b70a85 0x2e1b2138 0x4d2c6dfc 0x53380d13
0x650a7354 0x766a0abb 0x81c2c92e 0x92722c85 0xa2bfe8a1 0xa81a664b
```

*Block decomposition*

For each block, M $\in \{0, 1\}^{512}$, 64 words of 32 bits each ($W_i$) are constructed as follows:

- The first 16 are obtained by splitting M in 32-bit blocks

$$M = concat(W_1, W_2, \dots, W_{16})$$

- the remaining 48 are obtained with the formula:

$$W_i = \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16}, \qquad 17 \leq i \leq 64$$

*Hash computation*

First, eight variables are set to their initial values, given by the first 32 bits of the fractional part of the square roots of the first 8 prime numbers:

$H_1^{(0)} = 0x6a09e667 \quad H_2^{(0)} = 0xbb67ae85 \quad H_3^{(0)} = 0x3c6ef372 \quad H_4^{(0)} = 0xa54ff53a$

$H_5^{(0)} = 0x510e527f \quad H_6^{(0)} = 0x9b05688c \quad H_7^{(0)} = 0x1f83d9ab \quad H_8^{(0)} = 0x5be0cd19$

Next, the blocks $M(1), M(2), \dots, M(N)$ are processed one at a time:

for *t=1* to *N*

- construct the 64 blocks $W_i$ from $M(t)$, as explained above
- $set(a, b, c, d, e, f, g, h) = \left(H_1^{(t-1)}, H_2^{(t-1)}, H_3^{(t-1)}, H_4^{(t-1)}, H_5^{(t-1)}, H_6^{(t-1)}, H_7^{(t-1)}, H_8^{(t-1)}\right)$
- do 64 rounds consisting of:

$$T_1 = h + \Sigma_1(e) + Ch(e, f, g) + K_i + W_i$$

$$T_2 = \Sigma_0(a) + Maj(a, b, c)$$

$$h = g$$

$$g = f$$

$$f = e$$

$$e = d + T_1$$

$$d = c$$

$$c = b$$

$$b = a$$

$$a = T_1 + T_2$$

- compute the new value of $H_j^{(t)}$:

$$H_1^{(t)} = H_1^{(t-1)} + a$$

$$H_2^{(t)} = H_2^{(t-1)} + b$$

$$H_3^{(t)} = H_3^{(t-1)} + c$$

$$H_4^{(t)} = H_4^{(t-1)} + d$$

$$H_5^{(t)} = H_5^{(t-1)} + e$$

$$H_6^{(t)} = H_6^{(t-1)} + f$$

$$H_7^{(t)} = H_7^{(t-1)} + g$$

$$H_8^{(t)} = H_8^{(t-1)} + h$$

end for

The hash of the message is the concatenation of the variables $H_i^{(N)}$ after the last block has been processed

$$H = concat(H_1^{(N)}, \mathrm{H}_2^{(\mathrm{N})}, \mathrm{H}_3^{(\mathrm{N})}, \mathrm{H}_4^{(\mathrm{N})}, \mathrm{H}_5^{(\mathrm{N})}, \mathrm{H}_6^{(\mathrm{N})}, \mathrm{H}_7^{(\mathrm{N})}, \mathrm{H}_8^{(\mathrm{N})})$$

## SHA256 Hardware Implementation

The source code for the SHA256 digest implementation was based heavily on Brad Conte's software implementation (19), as well as work done by Jason Dahlstrom for process-

specific hardware security monitors (15). The digest is implemented using a single compound data structure and three functions:

```
typedef struct {
   uchar data[64];
   uint datalen;
   uint bitlen[2];
   uint state[8];
} SHA256_CTX;

void sha256_init(SHA256_CTX *ctx);
void sha256_update(SHA256_CTX *ctx, uchar data[], uint len);
void sha256_final(SHA256_CTX *ctx, uchar hash[]);
```

The function `sha256_init(SHA256_ctx *ctx)` is called to initialize the `SHA256_CTX` structure. The context is then passed in to one or more calls to `sha256_update()`, along with a buffer of data and its length, to generate/update the cumulative checksum (15).

```
void sha256_update(SHA256_CTX *ctx, uchar data[], uint len)
{
    // loop over each byte of input data and store in ctx->data
    for (i=0; i < len; ++i) {
        ctx->data[ctx->datalen] = data[i];
        ctx->datalen++;
        if (ctx->datalen == 64) { // if we have loaded all the data
            sha256_transform(ctx,ctx->data); // apply transform
            DBL_INT_ADD(ctx->bitlen[0],ctx->bitlen[1],512
            ctx->datalen = 0; // reset length
        }
    }
}
```

The `sha256_transform()` function applies the mathematical operation that compute the hash for a given block of data, as detailed earlier in this section.

```
void sha256_transform(SHA256_CTX *ctx, uchar data[])
{
    unsigned int a,b,c,d,e,f,g,h,i,j,t1,t2,m[64];

    for (i=0,j=0; i < 16; ++i, j += 4)
        m[i] = (data[j] << 24) | (data[j+1] << 16)
                               | (data[j+2] << 8) | (data[j+3]);
    for ( ; i < 64; ++i)
        m[i] = SIG1(m[i-2]) + m[i-7] + SIG0(m[i-15]) + m[i-16];
```

```
10
11      a = ctx->state[0];
12      b = ctx->state[1];
13      c = ctx->state[2];
14      d = ctx->state[3];
15      e = ctx->state[4];
16      f = ctx->state[5];
17      g = ctx->state[6];
18      h = ctx->state[7];
19
20      for (i = 0; i < 64; ++i) {
21          t1 = h + EP1(e) + CH(e,f,g) + k[i] + m[i];
22          t2 = EP0(a) + MAJ(a,b,c);
23          h = g;
24          g = f;
25          f = e;
26          e = d + t1;
27          d = c;
28          c = b;
29          b = a;
30          a = t1 + t2;
31      }
32
33      ctx->state[0] += a;
34      ctx->state[1] += b;
35      ctx->state[2] += c;
36      ctx->state[3] += d;
37      ctx->state[4] += e;
38      ctx->state[5] += f;
39      ctx->state[6] += g;
40      ctx->state[7] += h;
41  }
```

In the last step, the `sha256_final()` function performs a final padding step and an ultimate transform to complete the digest for the given block of data.

```
 1  void sha256_final(SHA256_CTX *ctx, uchar hash[])
 2  {
 3      unsigned int i = ctx->datalen;
 4
 5      // Pad whatever data is left in the buffer, and transform it
 6      if (ctx->datalen < 56) {
 7          ctx->data[i++] = 0x80;
 8          while (i < 56)
 9              ctx->data[i++] = 0x00;
10      }
11      else {
12          ctx->data[i++] = 0x80;
13          while (i < 64)
14              ctx->data[i++] = 0x00;
15          sha256_transform(ctx,ctx->data);
16          for ( i = 0; i < 56; i++ ) {
17              ctx->data[i] = 0x00;
18          }
```

```
    }

    // Append message's length in bits to the padding and transform.
    DBL_INT_ADD(ctx->bitlen[0],ctx->bitlen[1],ctx->datalen * 8);
    ctx->data[63] = ctx->bitlen[0];
    ctx->data[62] = ctx->bitlen[0] >> 8;
    ctx->data[61] = ctx->bitlen[0] >> 16;
    ctx->data[60] = ctx->bitlen[0] >> 24;
    ctx->data[59] = ctx->bitlen[1];
    ctx->data[58] = ctx->bitlen[1] >> 8;
    ctx->data[57] = ctx->bitlen[1] >> 16;
    ctx->data[56] = ctx->bitlen[1] >> 24;
    sha256_transform(ctx,ctx->data);

    // Since we use little endian (SHA uses big endian)we reverse
    // the bytes when copying the final state to the output
    for (i=0; i < 4; ++i) {
        hash[i]    = (ctx->state[0] >> (24-i*8)) & 0x000000ff;
        hash[i+4]  = (ctx->state[1] >> (24-i*8)) & 0x000000ff;
        hash[i+8]  = (ctx->state[2] >> (24-i*8)) & 0x000000ff;
        hash[i+12] = (ctx->state[3] >> (24-i*8)) & 0x000000ff;
        hash[i+16] = (ctx->state[4] >> (24-i*8)) & 0x000000ff;
        hash[i+20] = (ctx->state[5] >> (24-i*8)) & 0x000000ff;
        hash[i+24] = (ctx->state[6] >> (24-i*8)) & 0x000000ff;
        hash[i+28] = (ctx->state[7] >> (24-i*8)) & 0x000000ff;
    }
}
```

These functions are all tied together and called from within the top-level function `sha256()`. The function is detailed below, with some omissions for clarity:

```
void sha256( unsigned char data[MAXDATASIZE], // data to hash
       unsigned int  base_offset,          // offset of data to hash
       unsigned int  bytes,                 // length of data
       unsigned char digest[HASHSIZE] ) {   // output: digest

  // We work on buffers of up to 64 bytes
  unsigned char seg_buf[64]; // 64byte segment buffer
  unsigned int seg_offset = 0;//progress thru the region of interest
  int i=0;
  unsigned int n = bytes;

  // Initialize the SHA256 context
  SHA256_CTX sha256ctx;
  sha256_init(&sha256ctx);

  // Process the data (byte at a time...)
  while( n )
  {
    if( n >= 64 )
    {
      for (i=0; i<64; i++)
        seg_buf[i] = *(data + base_offset + seg_offset + i);
```

```
23        n -= 64;
24        seg_offset += 64;
25        sha256_update(&sha256ctx, seg_buf, 64);
26      }
27      else
28      {
29        for (i=0; i<n; i++)
30          seg_buf[i] = *(data + base_offset + seg_offset+i);
31        sha256_update(&sha256ctx, seg_buf, n);
32        n=0;
33      }
34    }
35
36    // Finish computing the hash and copy results back to proc mem
37    sha256_final(&sha256ctx, digest);
```

It is important to note that the `sha256_init`, `sha256_update()`, and `sha256_final()` functions are almost identical to the software implementation of the high-level algorithm description put forth at the beginning of this section, as the only hardware-specific changes that had to be made to the software implementation were the addition of the following interface directives:

```
#pragma HLS INTERFACE s_axilite port=data
#pragma HLS INTERFACE s_axilite port=digest
#pragma HLS INTERFACE s_axilite port=base_offset
#pragma HLS INTERFACE s_axilite port=bytes
#pragma HLS INTERFACE s_axilite port=return
```

These preprocessor directives specify the type of interface that the HLS compiler should infer for the top-level function during the *interface synthesis* step. Vivado HLS Interface Synthesis allows for the port interface to be automatically generated based on efficient industry standard interfaces. The developer need only specify the desired I/O protocol on the top-level function argument, and no further modifications to the source code are necessary.

### AES Overview

The Advanced Encryption Standard (AES) is a specification for the encryption of electronic data established by NIST in 2001 (20). AES is a *symmetric-key block cipher* algorithm, that is based on the Rijndael family of cipher algorithms. Symmetric-key ciphers are ciphers that use the same cryptographic keys for both encryption and decryption operations. A block cipher algorithm is a type of cipher that operates on a "block" of fixed-length data, with an unvarying transformation between plaintext and cipher text provided by a symmetric key. Block ciphers differ from stream ciphers in that the entire block of data is transformed at once, as opposed to a single bit at a time. AES is mandatory in several industry and commercial standards, including TLS, IPsec, secure shell network protocol (SSH), and the WPA2 Wi-Fi encryption standard (21). At the time of writing, there is no known practical (non-brute-force) attack against AES that would allow someone without knowledge of the key to read encrypted data. It is thus considered highly cryptographically secure (22).

### AES Algorithm Detail

The AES algorithm operates on a block size of 128 bits, and variants can use key lengths of 128, 192, and 256 bits. It is based on a substitution-permutation network, which combines both substitution and permutation of the data to ensure the algorithm's speed in both software and hardware (23). It should be noted that the internals of the AES algorithm involve Galois field arithmetic and other abstract mathematical topics that are beyond the scope of this thesis, and therefore will not be discussed in detail. The interested reader should refer to (22) for a comprehensive explanation of the mathematics.

AES uses a 4x4 column-major order matrix of bytes to represent the 128-bit (16-byte) input data block, referred to as the "state" matrix.

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix}$$

The state matrix is repetitively subjected to "rounds" of various mathematical transformations, each of which sequentially operate on all bytes of the state matrix until the final output is obtained. Iteration through the rounds is shown in Figure 6. The total number of rounds necessary to produce the result is dependent upon the number of bits in the key; a 256-bit key requires 14 rounds of repetition.

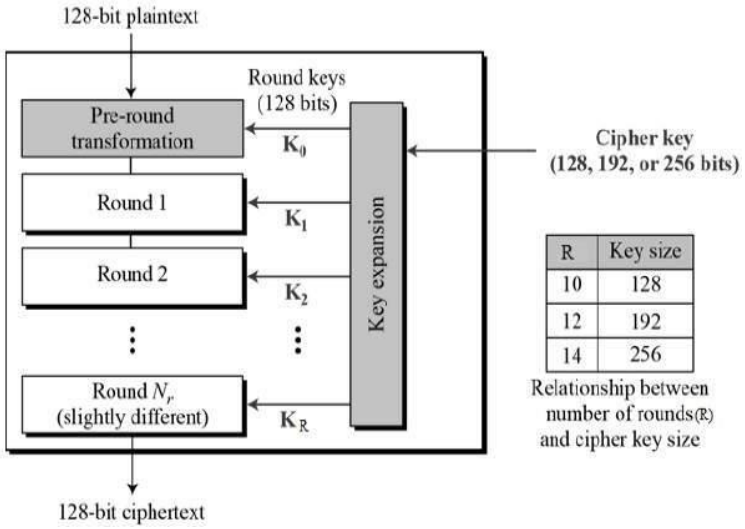


**Figure 6:** AES Algorithm Flow

Before the algorithm's rounds can begin, a set of *round keys* is derived from the original cipher using a procedure known as *Rijndael's key schedule*. The key schedule takes the original 256-bit cipher key, and expands it into a linear array of 128-bit round keys using a combination of bitwise operations and finite-field arithmetic. Once generated, the algorithm's rounds can begin.

The mathematical transformations performed on the state matrix in each round can be organized into sequential *stages*, sometimes referred to as *layers*, each of which operates upon all bits of the state matrix data at once. The algorithm begins with a key addition stage, followed by 13 rounds of four stages, followed by a 14th round of three stages. This applies for both encryption and decryption, with the exception that each stage of a round in the decryption algorithm is the inverse of its counterpart in the encryption algorithm (24). The four stages are defined as follows (21):

- **Key Addition:** the specific round key derived from the main key is XORed with each byte of the state to form the new transformed state. The addition operation in this layer is referred to as *AddRoundKey(state)*, and is visualized in Figure 7: Key Addition.

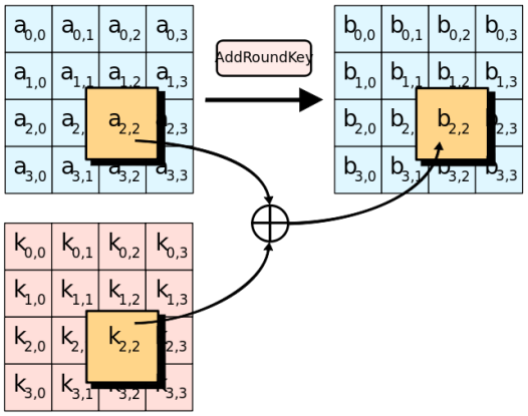


**Figure 7:** Key Addition

- **Byte Substitution:** Each element of the state is nonlinearly transformed using a lookup table with special mathematical properties called an S-box. The S-box is generated by determining the multiplicative inverse for a given number over the Galois Field $GF(2^8)$. Galois Fields and other related abstract algebraic topics are beyond the scope of this thesis and will not be explained, but it suffices to say that this stage introduces *confusion* to the data i.e., it assures that changes in individual state bits propagate quickly across the data path. This operation is referred to as *SubBytes(state)*in functional notation, and is shown in Figure 8.

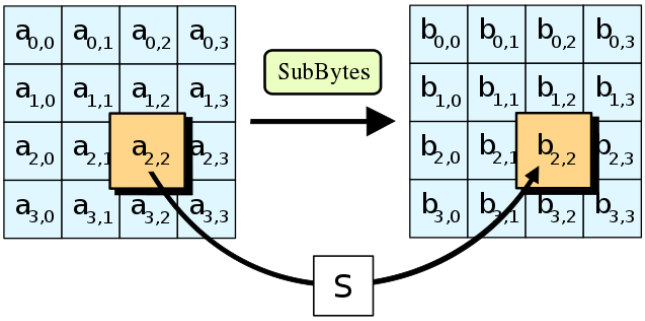


**Figure 8:** Byte Substitution

- **Diffusion:** Provides *diffusion* over all state bits. It consists of two sub-stages, both of which perform linear operations
    - *ShiftRows(state)* layer permutes the data on a byte-level, by performing a cyclical left shift to the bytes in each row. The number of shifts depends on the row. The ShiftRows operation is shown in Figure 9: ShiftRows.

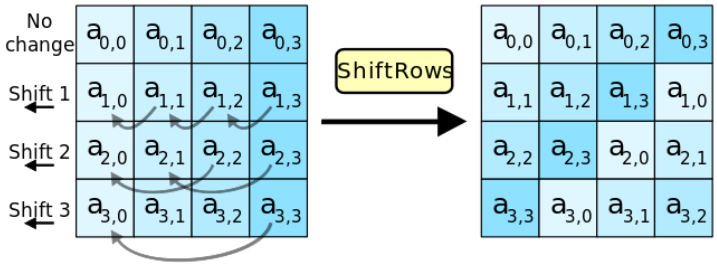


**Figure 9:** ShiftRows

- *MixColumns(state)* layer is a matrix operation which combines and mixes blocks of four bytes using an invertible linear transformation. This operation is shown in Figure 10.

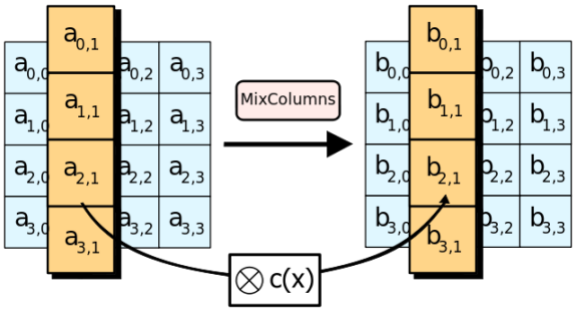


**Figure 10:** MixColumns

Each "round" (loop) through the algorithm consists of all three stages operating sequentially on the entire state matrix, exclusive of the first and last round, which differ such that encryption and decryption remain symmetric. Pseudocode describing the internal

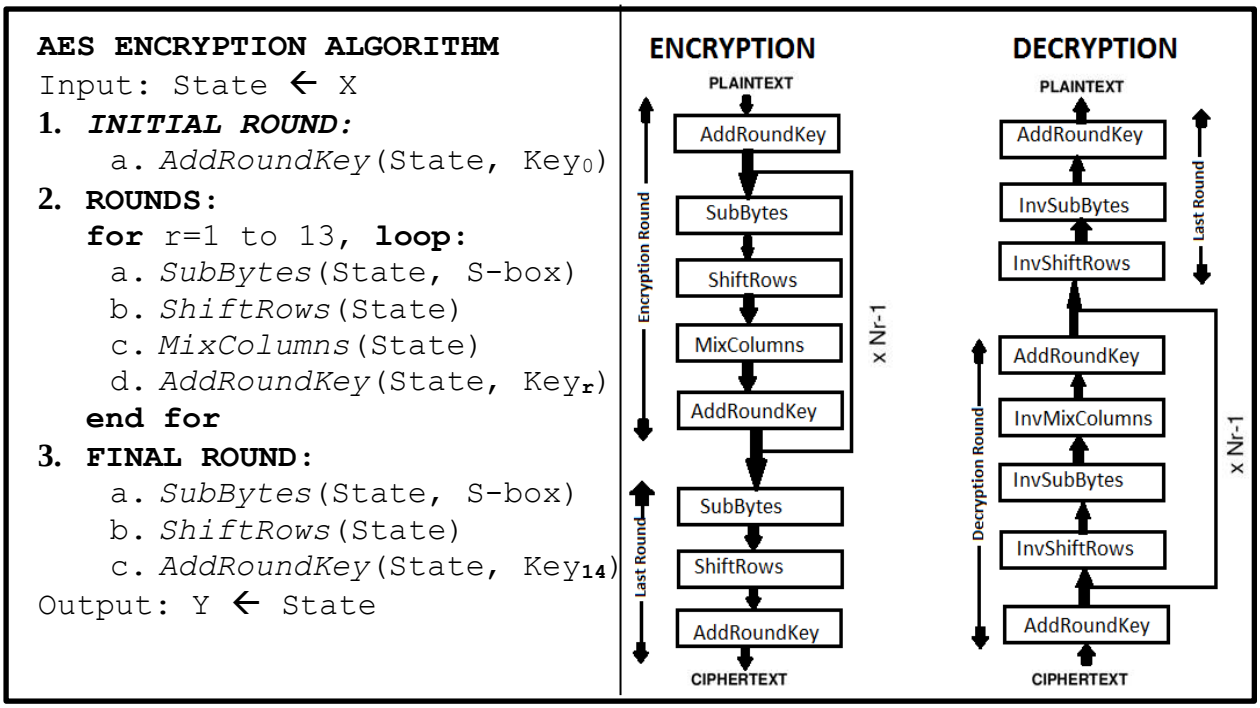


**Figure 11:** AES Algorithm and High-Level Flow

structure of the rounds for encryption, as well as a block diagram of the high-level algorithm flow is shown in Figure 11.

Like other symmetric block ciphers, AES specification supports both Encrypted Code Book (ECB) and Cipher Block Chaining (CBC) operating modes. The two modes specify different strategies for encrypting/decrypting data that is larger than the block size. ECB is the most basic mode of AES operation; it consists of running the algorithm exactly as described above on each block of input data with no modifications, and concatenating the results to form the output. In this case, the mapping between plaintext and cipher text is deterministic, and the same block of plaintext will always transform to the same cipher text. This is problematic because the presence of patterns in cipher text leaks valuable information about the plaintext. CBC mode eliminates this determinism by XORing the output of the first block with the input to the next, as shown in Figure 12. This makes the

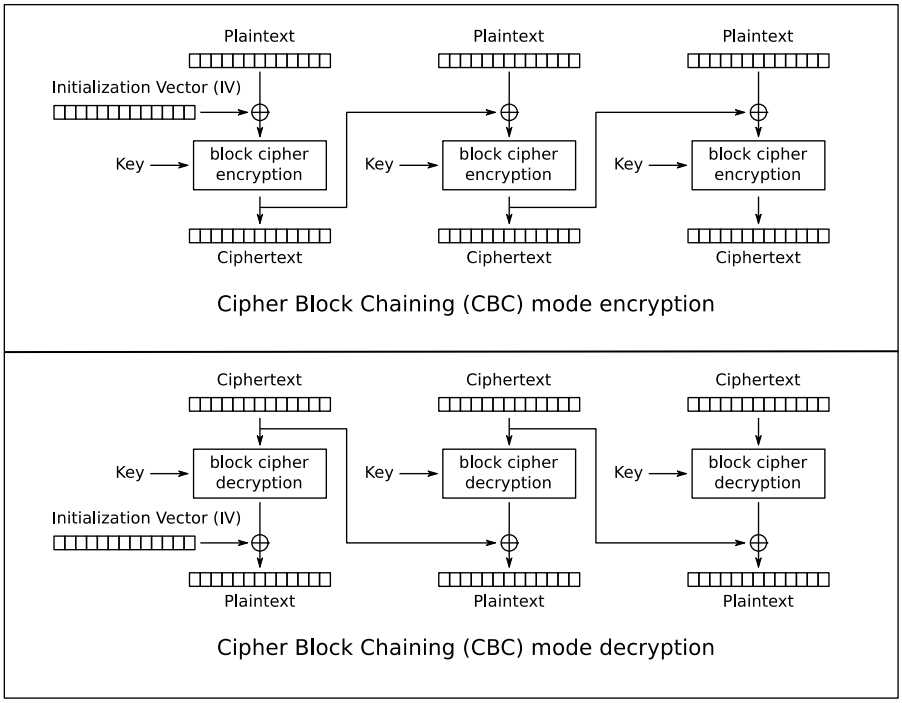


**Figure 12:** CBC Mode

output of each block dependent on the output of the previous block, thus two blocks of identical data at different places in the chain will always produce different results. The first block cipher input is XORed with an initialization vector, which must be used for both encryption and decryption.

### AES Hardware Implementation

The AES source code, like the SHA256 algorithm, was also based off Brad Conte's original software implementation (19) because of the emphasis on readability and clarity. The code utilizes a context structure to store global data, and operates on this structure via four core functions:

```
typedef struct {
   uint8_t key[32];
   uint8_t enckey[32];
   uint8_t deckey[32];
} aes_context;

void aes_init(aes_context *ctx, uint8_t * key);
void aes_encrypt_ecb(aes_context *ctx, uint8_t *plaintext);
void aes_decrypt_ecb(aes_context *ctx, uint8_t *ciphertext);
void aes_done(aes_context *ctx);
```

The `aes_init()` function initializes the context structure with the key, and performs the pre-round initialization detailed earlier in this section. The helper function `aes_expandEncKey()` is detailed in the full source code listings, found in appendix X.

```
void aes_init(aes_context *ctx, uint8_t *k)
{
    uint8_t rcon = 1;
    uint8_t i;

    for (i = 0; i < sizeof(ctx->key); i++)
        ctx->enckey[i] = ctx->deckey[i] = k[i];
    for (i = 8; --i;)
        aes_expandEncKey(ctx->deckey, &rcon);
}
```

`aes_encrypt_ecb()` encrypts a single block of data pointed to by their pointer arguments. It performs the full sequence of encryption "rounds" described earlier in this section through a series of calls to helper functions. These functions are detailed in appendix X. `aes_init()` must be called before either of these functions are used.

```
1  void aes_encrypt_ecb(aes_context *ctx, uint8_t *buf)
2  {
3      uint8_t rcon;
4      aes_addRoundKey_cpy(buf, ctx->enckey, ctx->key);
5      for (int i = 1, rcon = 1; i < 14; ++i)
6      {
7          aes_subBytes(buf);
8          aes_shiftRows(buf);
9          aes_mixColumns(buf);
10         if ( i & 1 )
11             aes_addRoundKey( buf, &ctx->key[16]);
12         else {
13             aes_expandEncKey(ctx->key, &rcon);
14             aes_addRoundKey(buf, ctx->key);
15         }
16     }
17     aes_subBytes(buf);
18     aes_shiftRows(buf);
19     aes_expandEncKey(ctx->key, &rcon);
21     aes_addRoundKey(buf, ctx->key);
22 }
```

`aes_decrypt_ecb()` decrypts a single block of data at the location pointed to by its argument. It also follows the decryption rounds described earlier in this section, and uses the same helper functions as `aes_encrypt_ecb()`, but in reverse order for decryption.

```
1  void aes_decrypt_ecb(aes_context *ctx, uint8_t *buf)
2  {
3      uint8_t rcon;
4      aes_addRoundKey_cpy(buf, ctx->deckey, ctx->key);
5      aes_shiftRows_inv(buf);
6      aes_subBytes_inv(buf);
7      for (int i = 14, rcon = 0x80; --i;) {
8          if ( ( i & 1 ) ) {
9              aes_expandDecKey(ctx->key, &rcon);
10             aes_addRoundKey(buf, &ctx->key[16]);
11         }
12         else
13             aes_addRoundKey(buf, ctx->key);
14         aes_mixColumns_inv(buf);
15         aes_shiftRows_inv(buf);
16         aes_subBytes_inv(buf);
17     }
```

```
18      aes_addRoundKey( buf, ctx->key);
19  }
```

These four core functions define the ECB operations, which only encrypt a single block of data at a time. They are called from the top-level function, `aescbc()`, which provides the interface and control logic for the HLS block, and implements the CBC operation for both encryption and decryption. The top-level function prototype is shown below:

```
void aescbc(ciphermode_t mode,
            uint8_t data_in[AESKEYSIZE],
            uint8_t data_out[AESBLKSIZE])
```

The two array arguments are the input and output data buffers, transferred from the PS to PL over the AXI bus. The function has five modes of operation, which are set via the mode argument. The modes are defined by the following enumerated type:

```
typedef enum {RESET=0, ENCRYPT, DECRYPT, SET_IV, SET_KEY} ciphermode_t;
```

The behavior of the block in each mode is determined by a switch statement in the top-level function. The skeleton code below shows the control flow of the top-level function, and details the important local variables.

```
 1 void aescbc(ciphermode_t mode,
 2             uint8_t data_in[AESKEYSIZE],
 3             uint8_t data_out[AESBLKSIZE])
 4 {
 5     uint8_t buf[AESBLKSIZE]; // crypto data from/to interface
 6     static uint8_t lastbuf[AESBLKSIZE]; // Used in decryption only
 7     static uint8_t iv[AESBLKSIZE]; // First XOR init vector
 8     static uint8_t xorv[AESBLKSIZE];// Current vector for next XOR
 9     static uint8_t key[AESKEYSIZE]; // Encryption key
10     static aes_context ctx;  // ECB block context
11
12     switch ( mode ) {
13     case RESET:
14     default:
15         // Outputs are reset, xorv is set to zero, aes_init() called
16         break;
17     case ENCRYPT:
18         // Encrypt data_in using CBC
19         break;
20     case DECRYPT:
21         // Decrypt data_in using CBC
22         break;
23     case SET_IV:
```

```
24          // Sets the IV array to data_in
25          break;
26      case SET_KEY:
27          // Sets the key array to data_in
28          break;
29      }
30  }
```

Both the SET_IV and SET_KEY modes allow the PS to load the iv[] and key[] arrays, respectively. This must be done before any encryption or decryption operations can be performed. In the default mode of RESET, data_out is initialized to zero, and the xorv array is initialized to the iv array. The xorv[] array holds the value that will be XORed with the input to each ECB operation for the CBC implementation in the ENCRYPT and DECRYPT modes. Recall that for CBC, the input to the first ECB operation must be XORed with an initialization vector, and each subsequent ECB output must be XORed with the next block of data before being processed as an input to the next ECB operation.

In ENCRYPT mode, the input data is first read in and XORed with the xorv[] array. Next, the data are encrypted using the aes_encrypt_ecb() function. The encrypted data is then stored into the xorv[] array for use in the next block cipher computation, and written to the data_out port. The implementation is shown below.

```
 1      case ENCRYPT:
 2          // copy data into buffer
 3          for (i=0; i<AESBLKSIZE; i++)
 4              buf[i] = data_in[i];
 5          // scramble input based on the iv/last cipher output block
 6          for (i=0; i<AESBLKSIZE; i++)
 7              buf[i] = buf[i]^xorv[i];
 8          // apply the ECB encryption algorithm
 9          aes_encrypt_ecb(&ctx, buf);
10          // copy the output to xorv for the next operation
11          for (i=0; i<AESBLKSIZE; i++)
12              xorv[i] = buf[i];
13          // copy the output to the destination region in memory
14          for (i=0; i<AESBLKSIZE; i++)
15              data_out[i] = buf[i];
```

```
16          break;
```

DECRYPT mode applies the same operations, but in a different order to facilitate CBC decryption. Here, the XOR operation must occur after the cipher operation rather than before, to properly "unscramble" the CBC steps. The Decryption mode is shown below.

```
 1      case DECRYPT:
 2           for(i=0; i<AESBLKSIZE; i++)
 3                buf[i] = data_in[i];
 4           // retain cipher block for next cycle's xorv[]
 5           for(i=0; i<AESBLKSIZE; i++)
 6                lastbuf[i] = buf[i];
 7           // apply the ECB decryption algorithm
 8           aes_decrypt_ecb(&ctx, buf);
 9           // unscramble results using the iv/last cipher block output
10           for(i=0; i<AESBLKSIZE; i++)
11                buf[i] = buf[i]^xorv[i];
12           // set up xorv for the next cycle
13           for(i=0; i<AESBLKSIZE; i++)
14                xorv[i] = lastbuf[i];
15           // copy the output to the destination region in memory
16           for(i=0; i<AESBLKSIZE;  i++)
17                data_out[i] = buf[i];
18           break;
```

**RSA Overview**

RSA is a *public key cryptography* algorithm created by Ron Rivest, Adi Shamir, and Leonard Adleman in 1978. *Public key cryptography*, also known as asymmetric cryptography, uses two different but mathematically linked keys, one public and one private (25). The public key can be shared with anyone, and is used to encrypt data. The private key must be kept secret, such that only the publisher of the private key is able to decrypt the original message. Public key cryptography is useful because it allows a user to send an encrypted message without a separate exchange of secret keys, differentiating it from symmetric key algorithms like AES. It also allows one to verify the authenticity of any message sent between the holder of the public key and private key. A message can be "signed" using a privately held decryption key, and anyone can verify this signature using

the corresponding public key. Signatures cannot be forged, and a signer cannot later deny the validity of a signature (26). Many protocols like SSH, OpenPGP, S/MIME, and SSL/TLS rely on RSA for encryption and digital signature functions. It is also used in software programs, such as internet browsers, which need to establish a secure connection over an insecure network like the Internet or validate a digital signature (25).

**RSA Algorithm Detail**

RSA is based on the principle of a *trapdoor function* – a function that is easy to compute in one direction, yet difficult to compute in the opposite direction without additional information. The trapdoor function within RSA involves calculating the solution to a modular exponentiation equation, and derives its security from the difficulty of factoring the product of two large primes. Multiplying two large primes together is a trivial task for computers, but the determining the original prime numbers from the sum is considered computationally infeasible for even the largest supercomputers in the world.

For example, it is practical to find three very large positive integers *e*, *d*, and *n* such that, with modular exponentiation for all integer m, the following two equalities hold:

$$(m^e)^d \equiv m \ (\text{modulo } n)$$

$$(m^d)^e \equiv m \ (\text{modulo } n)$$

However, given this relationship, it is computationally infeasible to find *d*, even with knowledge of *e, n,* and *m.* Given that appropriate values for *e, d,* and *n,* are selected, and *e* and *n* are made public, a message can be encrypted by performing the operation

$$E = m^e (\text{modulo } n)$$

where *m* is known as the *message* or *plaintext*, *E* as the *cipher text*, *e* as the *public exponent*, and *n* as the *modulus*. The cipher text *E* can later be decrypted using the operation

$$m = E^d (\text{modulo } n)$$

but only for the correct value of *d*, known as the *private exponent.* Therefore, the *public key* is defined as the combination of the *public exponent* and *modulus* <*e*,*n*>, and the *private key* is defined as of the combination of the *private exponent* and *modulus* <*d*,*n*>.

The RSA cryptosystem is commonly implemented using key lengths of either 1024, 2048, or 4096 bits. NIST recommends using a key length of 2048 bits for maximum security (27). The algorithm put forth in this thesis can accommodate keys of any length, which shall be discussed later in this section.

The RSA algorithm can be divided into four steps: key generation, encryption, decryption, and digital signing.

*Key Generation*

1. Choose two distinct and "random" prime numbers p and q, which are similar in magnitude but differ in length by only a few digits.
    - Prime integers can be efficiently found using many different algorithms, but most modern implementations use the Miller-Rabin primality test as specified in FIPS 186-4 (28).
2. Compute the modulus $n = pq$
3. Compute $\varphi(n) = (p - 1)(q - 1)$, where $\varphi$ is the Euler totient function. This value must be kept secret

4. Generate the public exponent *e*, selecting an integer *e* that satisfies $1 < e < \phi(n)$, and such that *e* and $\phi(n)$ are coprime; i.e. $\gcd(e, \phi(n)) == 1$.
5. Generate the private exponent *d*, such that $d \equiv e^{-1}(\text{modulo } \phi(n))$; i.e. *d* is the *modular multiplicative inverse* of the public exponent, modulo $\phi(n)$
6. The public key consists of the public exponent and the modulus, and the private key consists of the private exponent and the modulus.
    - Although $p, q,$ and $\phi(n)$ are not directly used in the operation of the RSA, they must be kept secret because they can be used to calculate *d* (26)

*Encryption*

Encrypting the plaintext *M* first involves transforming it into an integer *m*, such that $0 \leq m < n$. This is accomplished by applying a reversible padding scheme agreed upon by both parties to the plaintext, and encoding the resulting message characters in binary representation. This string of binary digits is the integer *m*, and is encrypted using the public key using the following equation.

$$E = m^e (\text{modulo } n)$$

The cipher text *E* can then be sent to the holder of the private key over a public channel, without the message contents being revealed.

*Decryption*

The holder of the private key can decrypt *E* by applying the same sequence of operations in reverse order. Plaintext *m* is obtained by applying the following equation to the cipher text

$$m = E^d \text{ (modulo } n)$$

and the original message *M* can be obtained by reversing the padding scheme applied to *m*.

*Digital Signing*

The fact that the encryption and decryption operations are inverses and operate on the same set of inputs also means that the operations can be employed in reverse order to obtain a digital signature scheme following the model originally put forth by Diffie and Hellman in 1976 (29). A plaintext message can be "digitally signed" by applying to it the same operation used to decrypt a block a block of cipher text:

$$s = \text{SIGN}(m) = m^d \pmod{n}$$

This signature can then be verified by applying the encryption operation to the signed message, and comparing the result with the original message text. This verifies that only the holder of the private key could have signed the message, but does require the original message to be transmitted along with the signed message for them to be compared. In practice, the plaintext *m* is generally some function of the message, for instance a formatted one-way hash of the message using an algorithm such as SHA256. This makes it possible to sign a message of any length with only one exponentiation, this verifying both the authenticity and integrity of a message.

**RSA Algorithm Implementation**

Unlike AES and SHA256, there are a variety of different algorithms and implementations that can implement the operations of the RSA cryptosystem. The mathematical operation at the core of the RSA algorithm is modular exponentiation, which is a computationally intensive task. Because RSA is used to encrypt session keys and hashes for the establishment and maintenance of a TLS session, it is imperative that fast and efficient modular exponentiation algorithms be used. There is a large body of research dedicated to

maximizing the performance of the modular exponentiation algorithm in both hardware and software for this reason.

*Fast Exponentiation*

The most efficient method of computing modular exponentiation is to use the classic "Square and Multiply" binary exponentiation algorithm to perform the exponentiation, with a few modifications to accommodate for the modulo operation. The method is based on the observation that, for a positive integer *n*:

$$x^n = \begin{cases} x(x^2)^{\frac{n-1}{2}}, & \text{if } n \text{ is odd} \\ (x^2)^{\frac{n}{2}}, & \text{if } n \text{ is even} \end{cases}$$

Combining this representation with binary decomposition (i.e. representing the base in terms of powers of two), the exponentiation can be reduced to successive squaring operations for each binary bit. For example, to compute $x^e$, where $e = 2^{16} + 1$, we do the following operations:

$$x^e = x^{2^{(16+1)}} = x \cdot 2^{16} = x \cdot x^{2 \cdot 2 \cdot \ldots \cdot 2} = x \cdot (((x^2)^2 \ldots)^2$$

which entails simply using repeated squaring to calculate $x^{2^{16}}$, and then multiplying this value by *x* to yield $x^e$ .

*Fast Modular Exponentiation*

The modular exponentiation operation can therefore be represented as simply an exponentiation operation, where the multiplication and squaring operations are done modulo some number. The exponentiation heuristics developed for computing some $x^e$ are also applicable for computing $x^e \pmod{n}$ (30).

The algorithm used for computing $m^e (\text{modulo } n)$ given the integers *m*, *e*, and *n* is the left-to-right (LR) Binary method, shown in Equation 1. The bits of the exponent are scanned from the most significant to the least significant, with a squaring operation performed for each bit, and a modular multiplication operation performed for each non-zero bit.

The algorithm functions by keeping a running accumulation of the square and multiply

**LR Binary Method**
*Input: m, e, n*
*Output:* E = m^e (modulo n), where $e = \sum_{i=0}^{k-1} e_i 2^i, \ \ e \in \{0,1\}$

1. $E = 1; \ P = m$
2. **for** *i=k-1* **to** *0*
   a. **if** $e_i == 1$ **then** $E = E * P$
   b. $P = P \cdot P \ (\text{modulo } n)$
3. **if** $e_{(k-1)} == 1$ **then** $E = E \cdot P \ (\text{modulo } n)$
4. **return** E

**Equation 1**

steps. At each stage, the (modulo *n*) operation ensures that the intermediate variables remain bounded by *n*. It is also possible to allow the intermediate variables to grow and perform the mod M as a single final operation, although this requires larger intermediate registers which is not usually desirable with modern key lengths (31). The LR binary method takes 2*n* operations in the worst case, and 1.5*n* operations on average (32). There exists no data dependency between modular squaring and multiplying, so both operations can theoretically be performed in parallel. However, the modulus operation requires a division operation, which has a large impact on the speed and area of the required hardware, and should be avoided wherever possible.

The naïve version of modular multiplication detailed above computes the multiplication first, and then applies a modular reduction step consisting of a division and *floor* operation. This approach has two essential drawbacks (31):

1. The word size is doubled for the intermediate result if the word sizes of the multiplication operands are equal (word size is calculated by $\log_2 a + \log_2 b$). With operand word sizes of up to 2048 bits, it is nearly impossible to build fast and small hardware multipliers
2. After each multiplication step, a modular reduction requiring integer division must be performed. This operation has a high hardware cost, and should be avoided.

An ingenious solution to this is to use *Montgomery modular exponentiation*, put forth by Peter L. Montgomery in his seminal 1985 paper. Montgomery modular exponentiation uses an operation called *Montgomery multiplication* to efficiently multiply two integers modulo *N* while avoiding the "trial division" by *N*. It is based around the principle that divisions can be converted into simple bitwise-shifts if multiplication is performed on the "*Montgomery residues"* of the integer operands.

*Montgomery Multiplication*

The algorithm for Montgomery multiplication computes the modular multiplication by transforming both integers into the Montgomery domain, performing the multiplication on the m-residues, and then finally transforming back into the integer domain. This allows exponentiation to be computed using the standard square and multiply algorithm, but with the multiplication steps performed in the Montgomery domain thus avoiding trial division.

Given *k*-bit integers *a*, *b*, and *n*, where $2^{k-1} \leq n < 2^k$ and radix $r = 2^k$, Montgomery multiplication provides an efficient method for computing $R = a \cdot b \pmod{n}$ through

the representation of the residue class modulo *n*. The Montgomery residue of an integer *a* < *n* with respect to *r*, known as the *n-residue,* is given by

$$\bar{a} = a \cdot r \text{ (modulo } n)$$

It can be shown that the set $\{i \cdot r \text{ (modulo } n) \mid 0 \leq i \leq n-1\}\, is$ a complete residue system; i.e. it contains all numbers between the range [0, *n*-1], and there is a one-to-one correspondence between the numbers in this range and the numbers in the aforementioned set (33). The Montgomery method leverages this property to introduce an efficient multiplication operation that computes the *n*-residue of the integers whose *n*-residues are given (30).

*Montgomery multiplication modulo n* is defined as the following operation

$$\bar{R} = \text{MM}(a, b, n) = a \cdot b \cdot r^{-1} \text{ (modulo } n)$$

where the result $\bar{R}$ is called the *Montgomery product.* The quantity $r^{-1}$ is the inverse of *r* modulo *n*, with the property

$$r^{-1} \cdot r = 1 \text{ (modulo } n)$$

This definition allows for the computation of an *n*-residue (i.e. transformation of an integer into the Montgomery domain) to be rewritten in terms of a Montgomery multiplication

$$\begin{aligned} \bar{x} &= x \cdot r \text{ (modulo } n) \\ &= x \cdot r^2 \cdot r^{-1} \text{ (modulo } n) \\ &= \text{MM}(x, r^2, n) \end{aligned}$$

The inverse transformation from the residue domain back to the integer domain can also be written in terms of Montgomery multiplication

$$\text{MM}(\bar{x}, 1, n) = \bar{x} \cdot 1 \cdot r^{-1} \text{ (modulo } n)$$

$$= x \cdot r \cdot 1 \cdot r^{-1} \text{ (modulo } n)$$

$$= x$$

With help of these two transformations, the modulo multiplication of two integers *a* and *b* can be performed. First, note that the Montgomery product of two *n*-residues $\bar{a}$ and $\bar{b}$ is also an *n*-residue, and can be written as follows

$$\bar{R} = \text{MM}(\bar{a}, \bar{b}, n) = \bar{a} \cdot \bar{b} \cdot r^{-1} \text{ (modulo } n)$$

which is also the *n*-residue of the product $R = a \cdot b$ (modulo $n$), since

$$\bar{R} = \bar{a} \cdot \bar{b} \cdot r^{-1} \text{ (modulo } n)$$

$$= a \cdot r \cdot b \cdot r \cdot r^{-1} \text{ (modulo } n)$$

$$= a \cdot b \cdot r \text{ (modulo } n)$$

Therefore, the multiplication of two integers *a* and *b* modulo *n*

$$c = a \cdot b \text{ (modulo } n)$$

can be defined purely using Montgomery multiplication

$$\bar{a} = \text{MM}(a, r^2, n) = a \cdot r \text{ (modulo } n)$$

$$\bar{b} = \text{MM}(b, r^2, n) = b \cdot r \text{ (modulo } n)$$

$$\bar{c} = \text{MM}\left(\bar{a}, \bar{b}, n\right) = \bar{a} \cdot \bar{b} \cdot r^{-1} \text{ (modulo } n)$$

$$= a \cdot r \cdot b \cdot r \cdot r^{-1} \text{ (modulo } n) = a \cdot b \cdot r \text{ (modulo } n)$$

$$c = \text{MM}(\bar{c}, 1, n) = \bar{c} \cdot 1 \cdot r^{-1} \text{ (modulo } n)$$

$$= a \cdot b \cdot r \cdot 1 \cdot r^{-1} \text{ (modulo } n) = ab \text{ (modulo } n)$$

It may not be immediately obvious that multiplying by $r^{-1}\,(\text{modulo}\,n)$ is an easier problem than simply multiplying by modulo *n*, however there is a simple radix-2 algorithm for doing so in the Montgomery domain that does not involve the trial division that the standard modulo operation requires (34). The algorithm is shown in Equation 2.

**Radix-2 Montgomery Multiplication (R2MM)**
*Input:* odd *N where* $N = \sum_{i=0}^{k-1} n_i 2^i,\ \ n \in \{0,1\},\ \ k = \lfloor \log_2 N \rfloor + 1,$
$X = \sum_{i=0}^{k-1} x_i 2^i,\ \ x \in \{0,1\},\ Y = \sum_{i=0}^{k-1} y_i 2^i,\ \ y \in \{0,1\}$
*Output:* S =MM(*A,B,N*) where $S = \sum_{i=0}^{k-1} s_i 2^i,\ \ s \in \{0,1\}$

1. $S[0] = 0$
2. **for** *i=0* **to** *k-1*
   a. $q_i = S[i] + x_i \cdot Y\ (\text{modulo } 2)$
   b. $S[i+1] = (S[i] + x_i \cdot Y + q_i \cdot N)/2$
3. **if** $(S[k] > N)$ **then** $S[k] = S[k] - N$
4. **return** S[*k*]

**Equation 2**

By using the Montgomery multiplication algorithm to transform integer operands into the Montgomery domain using $\bar{X} = MM(X, r^2, N) = X \cdot r\ (\text{modulo } N)$, trial division can be avoided. Additionally, $r^2(modulo\ n)$ can be precomputed to make the transformation more efficient, since cryptographic operations such as RSA change the modulus infrequently (34). Finally, the subtraction in step 3 can be avoided by a clever pre-multiplication of the input operands (35), and so is not necessary.

In summary, the algorithm computes the Montgomery product using only 2*k* *k*-bit additions and *k* one-bit right shifts, which is substantially simpler than conventional modular multiplication with division (34). Therefore, despite the initial conversion cost, if many Montgomery multiplications are followed by an inverse conversion, as in RSA, then a clear advantage over ordinary multiplication is obtained (36).

*Montgomery Modular Exponentiation*

Montgomery multiplication can now be used in a modified LR binary exponentiation algorithm to compute modular exponentiation more efficiently. This algorithm is shown in Equation 3.

**LR Montgomery Modular Exponentiation**
*Input: m, e, n*
*Output:* $x = m^e$ (modulo n), where $e = \sum_{i=0}^{k-1} e_i 2^i, \quad e \in \{0,1\}$

1. $\bar{m} = \text{MM}(m, r^2, n) = m \cdot r \text{ (modulo } n)$
2. $\bar{x} = \text{MM}(1, r^2, n) = r \text{ (modulo } n)$
3. **for** *i=k-1* **to** *0*
   a. $\bar{x} = \text{MM}(\bar{x}, \bar{x}, n)$
   b. **if** $e_i == 1$ **then** $\bar{x} = \text{MM}(\bar{m}, \bar{x}, n)$
4. $x = \text{MM}(\bar{x}, 1, n)$
5. **return** *x*
6. **return** E

**Equation 3**

The algorithm starts out by converting the base *m* to its Montgomery residue, and the result variable to the residue of 1. Recall that these transformations use values that can be precomputed, and thus only a multiplication is necessary. The inner loop performs the familiar *k* iterations of square (step 3a) and multiply (step 3b) operations from the LR binary method, but these operations are done modulo $2^k$ rather than modulo *n* since operands are in the Montgomery domain. Once the loop finishes, the integer domain representation *x* is obtained from the *n*-residue $\bar{x}$ via the final Montgomery multiplication by 1.

**RSA Hardware Implementation**

The hardware implementation of RSA is based on the Montgomery methods detailed above. It consists of three hierarchical functions: `rsa1024`, which handles the control

and configuration logic, `rsaModExp`, which performs the Montgomery modular exponentiation, and `montMult`, which performs the Montgomery multiplication.

The code snippet below shows the top level function, and associated state machine. The hardware core operates in three modes: `ENCRYPT`, where the `base` argument is the plaintext to be encrypted; `DECRYPT`, where `base` is the cipher text to be decrypted; and `DECRYPTKEYINIT`, where the private key buffer is loaded from `BRAM` hidden in the FPGA.

```
1  void rsa1024( memword_t privexp[NUM_MEMWORDS], // BRAM holding
2                  RSAmode_t mode,
3                  uintRSA_t base,     // base (plain/cipher)text
4                  uintRSA_t publexp,  // public exponent
5                  uintRSA_t modulus,  // modulus
6                  uintRSA_t *result ) // result
7  {
8
9     static uintRSA_t priv=0;
10
11    switch(mode)
12    {
13    // Load private key (from BRAM) into local buffer
14    case DECRYPTKEYINIT:
15          for (int i=0; i<NUM_MEMWORDS; i++)
16          {
17                priv.range(NUM_BITS-1,(NUM_BITS)-MEMWORD_SIZE)
                      = privexp[i];
18                if (i!=NUM_MEMWORDS-1)
19                      priv >>= MEMWORD_SIZE;
20          }
21          *result = 0;
22          break;
23
24    // Encrypts data using RSA modular exponentiation:
      // result = base^(private exponent) % modulus
25    // base is the plaintext, exponent is private exponent,
      // modulus is the shared modulus
26    case ENCRYPT:
27          rsaModExp(base,publexp,modulus,result);
28          break;
29
30    // Decrypts data using RSA modular exponentiation
31    // base is the ciphertext, exponent is private exponent,
      // modulus is the shared modulus
32    case DECRYPT:
33    default:
```

```
34          rsaModExp(base,priv,modulus,result);
35          break;
36
37     }
38 }
```

The code snippet below shows the modular exponentiation function. This function receives the base, exponent, modulus, and precomputed residue as function arguments from the top level. This allows a flexible implementation independent of the interface logic between the PL and the PS.

```
1  void rsaModExp(ap_uint<NUM_BITS+2> M,     // Base
2                 ap_uint<NUM_BITS+2> e,     // exponent
3                 ap_uint<NUM_BITS+2> n,     // modulus
4                 ap_uint<NUM_BITS+2> rbar, // r^2(n+2) mod n
5                 ap_uint<NUM_BITS+2> *out) // output
6  {
7    ap_uint<NUM_BITS+2> xbar=0, Mbar=0;
8    // Compute initial residues
9    montMult(rbar,M, n, &Mbar);
10   montMult(rbar,ap_uint<NUM_BITS+2>(1), n, &xbar);
11   // compute modular exponentiation using square+multiply algorithm
12   for (int i=NUM_BITS-1; i>=0; i--)
13   {
14     montMult(xbar,xbar,n,&xbar); // square
15     if (e.test(i)) {
16       montMult(Mbar,xbar,n,&xbar); // multiply
17     }
18   }
19   // undo montgomery transform
20   montMult(xbar,(ap_uint<NUM_BITS+2>)1,n,out);
21 }
```

The HLS function implementing Montgomery multiplication is as follows:

```
1  void montMult( ap_uint<NUM_BITS-1> X,
2                 ap_uint<NUM_BITS-1> Y,
3                 ap_uint<NUM_BITS-1> M,
4                 ap_uint<NUM_BITS-1>* outData)
5  {
6  #pragma HLS ALLOCATION instances=mul limit=256 operation
7    for (int i=0; i<NUM_BITS; i++) // Scan X bit-by-bit
8    {
9      if (X.test(i)) // if bit i of X is high, add S to Y
10       S += Y;
11     if (S.test(0)) // if resulting S is now even, add M
12       S += M;
13     S = S >> 1;    // Divide S by 2
```

```
14    }
15    *outData = S.range(NUM_BITS-1,0); // return result
16 }
```

The line beginning with `#pragma` instructs the HLS compiler to limit the number of instantiated hardware multipliers to 256, the maximum number available on the Zynq 7020. It is important to note that the HLS functions for Montgomery exponentiation and multiplication are nearly identical to the high-level description of the algorithm put forth in the previous section (Equation 2 and Equation 3). The complex nature of the RTL logic required to implement modular exponentiation is completely abstracted from the developer, allowing a succinct and readable implementation. This should be compared to a VHDL reference implementation of RSA from opencores.org, which requires almost 1,200 lines of code and only can support 512-bit keys. Performance comparisons and the tradeoffs of HLS vs HDL code will be discussed later in this chapter.

## Chapter 4: Operating System and OpenSSL integration

This chapter provides a detailed description of the kernel and application layers of the design, and explains how the hardware is integrated into the custom Linux image and OpenSSL software stack. The performance and security impact of hiding OpenSSL data structures in hardware will also be discussed.

Recall from chapter 1 that an operating system is necessary to provide the network stack, memory management, and other low-level support required by OpenSSL. A customized Linux distribution was created to serve as the target operating system, built using the Yocto Linux project development ecosystem. The creation of the customized Linux image using Yocto will be discussed later in this chapter, after detailing the Kernel and Application layers of the design.

**Kernel Layer**

The kernel layer comprises the loadable kernel module (LKM) device drivers that allow applications running in Linux user space to access the memory-mapped custom FPGA hardware detailed in the previous chapter. A loadable kernel module (LKM) is a piece of code that can be loaded and unloaded into the kernel upon demand, extending the functionality of the kernel without the need to reboot the system. LKMs are typically used to add support for new hardware, filesystems, or for adding system calls (37). A Kernel module is required to interface with hardware from inside a Linux user space program because Linux uses virtual memory, an idealized software abstraction of the storage resources that are actually available on a given machine (38). For a user space process to access a region of physical memory, such as the data and control registers of the hardware

algorithms put forth in this thesis, the physical addresses must be explicitly mapped into the process's virtual memory space. A LKM meant to interface with a physical device is often referred to as a "device driver". Figure 13 shows the location of kernel modules/device drivers within the overall operating system hierarchy.

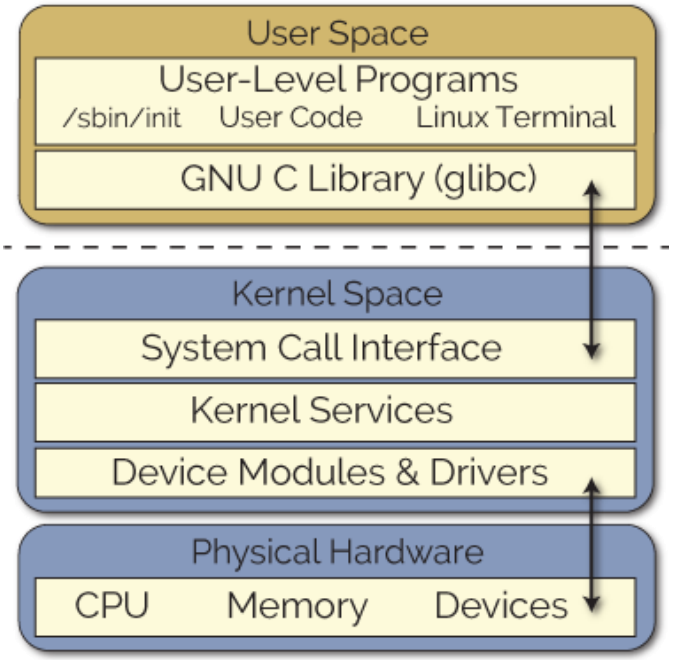


**Figure 13:** User-Kernel Space Hierarchy

*Basic Kernel Module Structure*

All Kernel modules must implement at least two functions: an initialization function which is called when the module is inserted into the kernel, and an exit (cleanup) function that is called just before it is removed (39). Modules are inserted and removed on the command line with the `insmod` and `rmmod` commands, respectively. Typically, the initialization function initializes and allocates memory for key data structures, registers a handler for an event within the kernel, or replaces one of the kernel functions with its own code. The exit function is used to undo whatever the initialization function does, so the module can be unloaded safely without risk of the base kernel crashing (39). Initialization and exit

functions are declared through the `module_init()` and `module_exit()` macros, respectively. These macros are defined in the kernel include file `<linux/init.h>`, and register the module's init and exit functions with the kernel. An example "hello world" example shows their usage:

```
 1 #include <linux/module.h>  /* Needed by all modules */
 2 #include <linux/kernel.h>  /* Needed for KERN_INFO */
 3 #include <linux/init.h>    /* Needed for init/exit macros */
 4
 5 static int hello_data __initdata = 0123456789;
 6
 7 static int __init hello_init(void)
 8 {
 9    printk(KERN_INFO "Hello, world %d\n", hello_data);
10    return 0;
11 }
12
13 static void __exit hello_exit(void)
14 {
15    printk(KERN_INFO "Goodbye, world!\n");
16 }
17
18 module_init(hello_init);
19 module_exit(hello_exit);
```

The `__init` macro means that for a built-in driver (i.e. compiled into the kernel, not a LKM) the function is only used at kernel initialization and that it can be discarded and its memory freed up after that point. Likewise, the `__exit` macro notifies the kernel that if the code is used for a build-in driver, the function is not required, since it cannot be unloaded (40). Also notice the use of the `printk()` function to display data to the console, rather than the usual `printf()` function used in user space C programs. This is because kernel code cannot access the standard libraries of code written for user space programs, such as `<stdio.h>`. Kernel code resides and executes in kernel space, which only has access to symbols defined by the running kernel. The kernel does provide a logging mechanism to log messages or warnings to the kernel message buffer through the

`printk()` macro, which has roughly the same syntax as its user space counterpart `printf()`.

*The Kernel and Devices*

A kernel module implements some specific functionality provided by an exposed kernel API, depending on the purpose of the module. The LKMs developed in this thesis are device drivers, and therefore must adhere to the rules of how Linux chooses to represent devices: as *files*. The often-quoted adage "*Everything is a file*" describes one of the defining features of Unix, Linux, and its various derivatives — that the range of input/output resources such as documents, directories, hard-drives, modems, keyboards, printers and even inter-process and network communications are simple streams of bytes exposed through the filesystem name space (41). Therefore, a Linux device driver must re-implement the kernel-level file operations API, `file_operations`, to give a user space program the ability to open, close, read, and write two/from a device.

The Linux kernel groups devices into two categories, character devices and block devices. A character device typically transfers data to and from a user application — they behave like pipes or serial ports, instantly reading or writing the byte data in a character-by-character stream. A block device behaves in a similar fashion to regular files, allowing a buffered array of cached data to be viewed or manipulated with operations such as reads, writes, and seeks (40). Both device types can be accessed through device files that are attached to the file system tree, and require a driver to create the device file mapping and re-implement the kernel `file_operations` data structure to allow user space access through system calls. The `file_operations` structure is very large and so will not be

shown in its entirety, however the fields that are re-implemented will be described later in this chapter.

This thesis uses the character device abstraction due to its simplicity and support from the Xilinx software ecosystem. The basic structure of the LKMs created for the SHA256, AES, and RSA algorithms are essentially the same, with minor changes to accommodate the specific requirements of the implemented digest/cipher. Given the similarities, this paper uses the SHA256 kernel module as an exemplar for describing the implementation of all three kernel modules, and mentions any key differences as they are introduced. The reader is referred to the source code of the AES and RSA modules for further specific details.

*Kernel Module Implementation*

The first part of each module is the inclusion of the appropriate kernel headers necessary to extend the kernels functionality. The includes for each of the LKMs are as follows:

```
1 #include <linux/init.h>   // __init and __exit macros
2 #include <linux/module.h> // Core header for loading LKM into kernel
3 #include <linux/device.h> // Support for kernel Driver Model
4 #include <linux/kernel.h> // Types, macros, functions for the kernel
5 #include <linux/fs.h>     // Linux file system support
6 #include <asm/uaccess.h>  // Required for the copy to user function
7 #include <linux/io.h>     // Required for printk()
8 #include <linux/sizes.h>  // Standard kernel data sizes
```

Next, each LKM defines the base address and register offsets for each of the hardware control registers and top level ports on the AXI bus. The register offsets are generated by HLS, and the physical base address is assigned by Vivado in the block designer. The defines specific to the SHA256 LKM are shown below:

```
1 // Device-specific Register (PHYSICAL) Addresses
2 #define SHA256BASEADDR 0x43C10000
3 #define XSHA256_AXILITES_ADDR_AP_CTRL          0x000
4 #define XSHA256_AXILITES_ADDR_GIE              0x004
5 #define XSHA256_AXILITES_ADDR_IER              0x008
6 #define XSHA256_AXILITES_ADDR_ISR              0x00c
7 #define XSHA256_AXILITES_ADDR_BASE_OFFSET_DATA 0x200
8 #define XSHA256_AXILITES_BITS_BASE_OFFSET_DATA 32
```

```
 9 #define XSHA256_AXILITES_ADDR_BYTES_DATA       0x208
10 #define XSHA256_AXILITES_BITS_BYTES_DATA       32
11 #define XSHA256_AXILITES_ADDR_DATA_BASE        0x100
12 #define XSHA256_AXILITES_ADDR_DATA_HIGH        0x1ff
13 #define XSHA256_AXILITES_WIDTH_DATA            8
14 #define XSHA256_AXILITES_DEPTH_DATA            256
15 #define XSHA256_AXILITES_ADDR_DIGEST_BASE      0x220
16 #define XSHA256_AXILITES_ADDR_DIGEST_HIGH      0x23f
17 #define XSHA256_AXILITES_WIDTH_DIGEST          8
18 #define XSHA256_AXILITES_DEPTH_DIGEST          32
```

These addresses are necessary to include in the kernel module because the entire range of memory corresponding to the register addresses are mapped into virtual memory inside the LKM initialization function. Next, the device name and device class name are defined as follows:

```
#define DEVICE_NAME "sha256char" //device appears at /dev/DEVICE_NAME
#define CLASS_NAME "sha256" // internal class reference for driver
```

The device name is the string label that appears in the `/dev` directory when the character device is created, while the class name is a string name for the device class object registered with the kernel. Introduced in kernel version 2.3, device classes provide a high-level abstraction of a device that ignore the low-level implementation details, allowing user space programs to work with devices based on what they do, rather than how they are connected or how they work (42). Both parameters are used in the LKM initialization function. A series of additional operations required to register the module with the kernel are also performed at this point in the module, however they are omitted for brevity.

Next, an important series of module-global variables pertaining to the registry of the module with the kernel are declared:

```
1 // Driver-specific registry info
2 static int majorNumber;  // Stores device number
3 static struct class* sha256charClass = NULL;//device class ptr
4 static struct device* sha256charDevice = NULL;//device struct ptr
6 // pointer to virtual memory address for the device
7 static void __iomem *vbaseaddr = NULL;
```

`majorNumber` holds the major device number for this driver module. When a program requests access to a device file, the major number specifies which device driver is called to perform the requested input/output operation. `sha256charClass` is a pointer to a `struct class*` that will be created by the kernel for the device in the LKM initialization function. Similarly, `sha256charDevice` is a pointer to a `struct device*` that is set immediately after in the initialization function. Both structure initialization operations will be discussed when the initialization function is described later in this section. The pointer `void __iomem *vbaseaddr` is also declared, which will hold the base address of the device registers in virtual memory once it is mapped inside the initialization function. The module initialization function is shown below, with omissions for clarity and length:

```
1  static int __init sha256_init(void)
2  {
3      // request physical memory for driver
4      if (!request_mem_region(sha256BASEADDR, SZ_64K, "sha256")) {
5          printk(KERN_ALERT "Failed to request memory region\n");
6          return -EBUSY;
7      }
8      // map reserved physical memory into into virtual memory
9      vbaseaddr = ioremap(sha256BASEADDR, SZ_64K);
10     if (! vbaseaddr) {
11         printk(KERN_ALERT "Unable to map virual memory\n");
12         release_mem_region(sha256BASEADDR, SZ_64K);
13         return -EBUSY;
14     }
15
16     // Try to dynamically allocate a major number for the device
17     majorNumber = register_chrdev(0, DEVICE_NAME, &fops);
18     if (majorNumber<0){
19         printk(KERN_ALERT "Failed to register a major number\n");
20         return majorNumber;
21     }
22
23     // Register the device class
24     sha256charClass = class_create(THIS_MODULE, CLASS_NAME);
25     if (IS_ERR(sha256charClass)){
26         unregister_chrdev(majorNumber, DEVICE_NAME);
27         printk(KERN_ALERT " Failed to register device class\n");
28         return PTR_ERR(sha256charClass);
```

```
29      }
30
31      // Register the device driver
32      sha256charDevice = device_create(sha256charClass, NULL,
33                                MKDEV(majorNumber, 0), NULL, DEVICE_NAME);
34      if (IS_ERR(sha256charDevice)){
35          class_destroy(sha256charClass);
36          unregister_chrdev(majorNumber, DEVICE_NAME);
37          printk(KERN_ALERT "Failed to create the device\n");
38          return PTR_ERR(sha256charDevice);
39      }
40
41      // initialize hardware parameters
42      iowrite32(SHA256_MSG_SIZE,
                  vbaseaddr+XSHA256_AXILITES_ADDR_BYTES_DATA);
43      iowrite32(0, vbaseaddr+XSHA256_AXILITES_ADDR_BASE_OFFSET_DATA);
44
45      return SUCCESS;
46  }
```

Notice how the global variables described earlier in this section (`vbaseAddr`, `majorNumber`, `sha256charDevice`, etc.) are each set using the return values from built-in kernel functions. This ensures that all the necessary structures representing devices within the kernel are populated and registered before the module can do any real "work". At the end of the function, two calls to `iowrite32` set the initial values for the hardware AXI data registers. In the case of SHA256, this operation zeros out the data register, and populates the `base_offset` register – both of which were defined in the top-level HLS function.

The LKM exit function servers the inverse purpose as the initialization function, undoing the memory mapping and device registration. It is called when the LKM is unloaded using the `rmmod` command. If kernel memory were allocated within the LKM, this function should free it. The exit function is shown below:

```
1 static void __exit sha256_exit(void) {
2      iounmap(vbaseaddr); // unmap device IO memory
3      // free the reserved kernel memory
4      release_mem_region(sha256BASEADDR, SZ_64K);
5      // remove the device
```

```
 6     device_destroy(sha256charClass, MKDEV(majorNumber, 0));
 7     // unregister the device class
 8     class_unregister(sha256charClass);
 9     // remove the device class
10     class_destroy(sha256charClass);
11     // unregister the major number
12     unregister_chrdev(majorNumber, DEVICE_NAME);
13 }
```

The two remaining functions from the `file_operations` structure, `write()` and `read()`, are where the data transfers two and from the hardware occur, respectively. Once the device file is opened, data can be read/written to it from the calling user space program through the re-implemented functions. Recall that only the kernel module is aware of the device's true location in both physical and virtual memory. This abstracts all implementation details from any calling user space program. The `write()` function is shown below

```
 1 static ssize_t sha256_write(struct file *filep, const char *buffer,
                               size_t len, loff_t *offset)
 2 {
 3     // init hardware parameters: set bytes=0 and base_offset=0
 4     iowrite32(len, vbaseaddr+XSHA256_AXILITES_ADDR_BYTES_DATA);
 5     iowrite32(0, vbaseaddr+XSHA256_AXILITES_ADDR_BASE_OFFSET_DATA);
 6
 7     // copy len bytes of data from userspace into kernel msg buffer
 8     copy_from_user((char*)message, buffer, len);
 9
10     // write data from kernel msg buffer to PL register region
11     memcpy_toio(vbaseaddr+XSHA256_AXILITES_ADDR_DATA_BASE,
                   (char*)message, SHA256_MSG_SIZE);
12
13     // start AES block using read-modify-write on ap_ctrl register
14     sha256_runonce_blocking();
15
16     return len;
17 }
```

The write function first initializes the HLS block's `bytes` and `base_offset` registers, detailed in chapter 3. It then copies `SHA256_MSG_SIZE` bytes from the user space buffer to the virtual memory region corresponding to the HLS block's `data` register. Finally, a call to `sha256_runonce_blocking()` sets the HLS block `ap_start` signal high to

begin the digest computation, and then polls the `ap_done` register until the hardware operation is complete.

```
1 static void sha256_runonce_blocking(void)
2 {
3     unsigned int ctrl_reg;
4
5     // set ap_start high using read-modify-write
6     ctrl_reg = ioread32(vbaseaddr+XSHA256_AXILITES_ADDR_AP_CTRL)
                          &0x80;
7     iowrite32(ctrl_reg | 0x01,
                vbaseaddr + XSHA256_AXILITES_ADDR_AP_CTRL);
8
9     // wait for completion
10    while(!(ioread32(vbaseaddr + XSHA256_AXILITES_ADDR_AP_CTRL) >>1)
           & 0x1) {;};
11 }
```

In a more fully-featured system it would be wise to replace the blocking write with a non-blocking write (using a mutex or other synchronization primitive) so other work can be done while the hardware is busy. However, for the purposes of this thesis, a simple blocking write is sufficient.

The `read()` function simply copies the data in the HLS block data register back into the user space buffer. There is no need for any register polling, since the `write()` function will not return until hardware computation has completed.

```
1 static ssize_t sha256_read(struct file *filep, char *buffer,
                             size_t len, loff_t *offset)
2 {
3     // Read digest from PL data register
4     memcpy_fromio((char*)digest,
                    vbaseaddr+XSHA256_AXILITES_ADDR_DIGEST_BASE,
                    SHA256_DGST_SIZE);
5
6     // Copy digest back into userspace (*to,*from,size)
7     copy_to_user(buffer, (char*)digest, len);
8     return len;  // return # of bytes read
9 }
```

It should be mentioned again that, although this section used the SHA256 module as an example, both the AES and RSA kernel modules use the same general structure. Outside

of the minutiae of how input/output data is defined and operated upon, the only key difference between AES and RSA LKMs is their use of the `ioctl()` operation. The `ioctl` operation, an abbreviation of Input/Output ConTroL, is a system call for device-specific input/output operations and other operations which cannot be expressed by regular system calls (43). An example of a device specific operation would be setting the baud rate on a serial port, or instructing an external disc to eject its physical media. The structure of the AES and RSA hardware modules require setting an operating mode (encrypt/decrypt), which is implemented through the `ioctl()` interface. The file-specific `ioctl` function is exposed as another member of the kernel `file_operations` structure, with the following prototype:

```
// Called by the standard ioctl syscall, when invoked on a device file
1 long (*unlocked_ioctl) (struct file *, unsigned int ioctl_num,
                          unsigned long ioctl_param);
```

The `ioctl_num` argument allows the user space application to specify the device-specific operation that it wishes to perform. These numbers are defined by the device driver, and must be registered with the kernel and should be defined in a common header. The `ioctl_param` argument is an optional parameter allowing data to be passed into the `ioctl` function from user space. It should be noted that use of the `ioctl` operation prohibits the use of dynamic device number registration because the `ioctl` numbers registered with the kernel need to be associated with a device major number at module insertion time. Therefore, a static major number needs to be defined before the module is loaded into the kernel, defined in the LKM header file as follows:

```
1 #define IOCTL_SET_MODE _IOR(MAJOR_NUM, 0, char)
2 #define IOCTL_GET_MODE _IOR(MAJOR_NUM, 1, char)
```

The first arguments to the `IOCTL_SET_MODE` kernel macro is the major device number of the LKM. The second argument is the ioctl number, as there could be many for a given module. The third argument is the data type that the calling process desires to transport to/from the kernel in the `ioctl_param` argument. For AES and RSA, the `ioctl` operation allows the mode argument to be set in hardware from user space before transferring actual data via a read/write. It also allows a user space application to query the current mode of the hardware. The `ioctl` operation for AES is defined as follows:

```
1  static long aes_ioctl(struct file *file, unsigned int ioctl_num,
2                        unsigned long ioctl_param) {
3      int retval = 0;
4      unsigned int ctrl_reg = 0;
5
6      // Switch according to the ioctl called
7      switch (ioctl_num) {
8      case IOCTL_SET_MODE:
9          // get mode parameter from user if valid, and write to PL
10         if ( (mode >= 0) && (mode <= 4)) {
11             mode = (ciphermode_t)ioctl_param;
12             iowrite8(mode,
                        vbaseaddr+XAESCBC_AXILITES_ADDR_MODE_DATA);
13
14             // if mode==RESET, manually start hw since there will be
15             // no pending write() call to to register new value
16             if (mode == RESET)
17                 wsaes_runonce_blocking();
18         } else  {
19             printk(KERN_INFO "IOCTL_SET_MODE: INVALID MODE\n");
20             retval = -EINVAL;
21         }
22         break;
23
24     case IOCTL_GET_MODE:
25         // copy mode from hw reg back to userspace
26         retval = put_user(mode, (ciphermode_t*)ioctl_param);
27         break;
28
29     default:
30         printk(KERN_INFO "ERROR, IMPROMER IOCTL NUMBER <%d>\n",
                       ioctl_num);
31         retval = -ENOTTY;
32         break;
33     }
34     return retval;
35 }
```

An example of an invocation of the AES `ioctl` from user space is as follows:

```
1 int ret = ioctl(fd, IOCTL_SET_MODE, RESET); //set mode of hw to RESET
2 // ... do things here ...
3 ret = ioctl(fd, IOCTL_GET_MODE, &mode); // get mode from hw
```

*Kernel Layer Issues with RSA*

Both the SHA256 and AES kernel modules were successfully implemented in the design, and their results are detailed in Chapter 5, however the RSA kernel module was never able to be successfully integrated due to a likely OS memory management bug with the Xilinx AXI drivers for Linux. The RSA HLS block was verified in hardware when running on a bare metal system, but was unable to function when running under Linux. This esoteric bug was brought to the attention of Xilinx, who were unable to mediate the issue within a suitable timetable for this thesis. The bug investigation is still ongoing at the time of writing.

The bug revealed itself when unit testing the kernel module. Reads from the output result register consistently returned random values, despite a logic analyzer IP core verifying that the input data registers were set correctly. The logic analyzer was also used to verify that there was no data corruption occurring in between the output data register and the AXI interconnect. To eliminate the possibility of a hidden bug in the kernel module, the Linux utility `devmem` was used to manually load the input data registers and read from the output data register. The same behavior was observed, prompting a bug report to be filed with Xilinx. To supplement the Xilinx investigation, a program using the /dev/mem kernel drivers supplied by Xilinx was created test the hardware core; this test also yielded the same result. The unofficial explanation for the bug provided by a Xilinx community forum moderator was that "*as Linux boots up, it's [most likely] doing something that breaks the HLS blocks. HLS state machines do seem to be somewhat "fragile" - maybe Linux is issuing*

*a reset while an AXI transaction is in progress (or something similar) and that's leaving the HLS block in a state that it can't recover from*" (44).

Because this bug prevented the RSA core LKM from functioning correctly, the RSA algorithm's kernel and application layer performance could not be evaluated alongside of SHA and AES. Performance data was still collected for the RSA hardware layer, and is analyzed in the next chapter.

**Application Layer**

Recall from chapter 1 that the application layer contains all remaining software required to integrate the hardware cryptographic blocks into OpenSSL. This consists of two additional components for each cryptographic function: a user space API, and an OpenSSL engine.

*User Space API*

The user space APIs abstract the low-level details of interacting with the LKM, providing a more logical and intuitive set of functions to control the hardware instead of using the standard Linux device file operations (open, close, read, write, ioctl). They are not necessary, but greatly simplify user space interaction with the kernel module. The API for each LKM contains only two functions: an initialization function, and the core digest/cipher function, both of which shall be explored using the AES operation as an exemplar. The initialization function simply checks whether the LKM for the associated algorithm is loaded into the running kernel. It does this by checking for the presence of the device file in the `/dev` directory.

```
1 int32_t aes256init(void) {
2     // Check if we can access the device file
3     if ( access( devicefname, F_OK ) != -1 ) {
4         printf("Found device!\n");
```

```
5         return 0;
6     } else {
7         fprintf(stderr, "ERROR: Couldn't find device %s\n",
                  devicefname);
8         return -1;
9     }
10 }
```

The core digest/cipher function can set the operation mode, as well as initiate the encryption/decryption operation in hardware. Skeleton code for the function is shown piecemeal below, with all error checking, argument testing, and informational messages removed for clarity:

```
1 int32_t aes256(int mode, uint8_t *inp, uint32_t inlen,
                 uint8_t *outp, uint32_t *lenp) {
2     int32_t fd, ret;
3     // Open the device with read/write access
4     fd = open("/dev/wsaeschar", O_RDWR);
5     // Reset block
6     ret = ioctl(fd, IOCTL_SET_MODE, RESET);
7     // Set mode to ENCRYPT or DECRYPT
8     ret = ioctl(fd, IOCTL_SET_MODE, (ciphermode_t)mode);
```

First, the device file is opened with read/write positions, the block is reset, and the block is set to the mode passed in the `mode` argument.

```
10    int orignumbytes; //original number of bytes in the input data
11    uint8_t lastblock[AESBLKSIZE];//last block to send if encrypting
12
13    // if we are encrypting, we must deal with padding separately
14    if (mode == ENCRYPT) {
          //number of data and padding bytes in the last block
15        int modlen = inlen % AESBLKSIZE;// last data bytes
16        int numpadbytes = AESBLKSIZE-modlen; // last padding bytes
17        //set output length to next non-zero multiple of block size
18        *lenp = inlen + numpadbytes;
19        // loop boundary for looping through the blocks
20        orignumbytes = *lenp - AESBLKSIZE;
21        // Construct "last block" of data to send, composed of the
22        // remaining bytes that don't fit into the 16-byte block
23        for (int i=0; i<AESBLKSIZE; i++)
24            lastblock[i] = (i < modlen) ?
                              inp[orignumbytes + i] : numpadbytes;
25    } else {
26        // we are not encrypting, so don't need to pad data
27        *lenp = inlen;
28        orignumbytes = inlen;
29    }
```

The lines above manage the block-level padding for the data if encryption is desired. The padding scheme implemented is known as PKCS#7, and will not be discussed here.

```
30      // initialize output memory to all zeros
31      memset((void*)outp,0,*lenp);
32
33      // MAIN DATA SENDING LOOP:
34      // send each complete 16-byte block of data to the LKM
35      for (int i=0; i<orignumbytes; i+=AESBLKSIZE) {
36          // send 16 byte block from caller to AES block
37          ret = write(fd, &(inp[i]), AESBLKSIZE);
38          // read back processed 16 byte block into caller memory
39          ret = read(fd, &(outp[i]), AESBLKSIZE);
40      }
```

Following the padding operation, the main data sending loop sequentially writes blocks of data to hardware through the LKM, and reads back and stores each result. The hardware by default works in CBC mode, however ECB mode can be used by simply resetting the block each time through the loop. Once out of the main data loop, an ultimate block of partially or fully padded data must also be sent if in encryption mode, such that padding is compliant with the PKCS#7 standard.

```
41      // if we are encrypting, deal with the extra padding bytes
42      if (mode == ENCRYPT) {
43          // send final padded block
44          ret = write(fd, lastblock, AESBLKSIZE);
45          // read back processed final padded block
46          ret = read(fd, &(outp[orignumbytes]), AESBLKSIZE);
47      }
48      // close and exit
49      close(fd)
50      return SUCCESS;
51  }
```

The API for both SHA and RSA are not detailed for the sake of brevity, as they use the same structure as AES. The curious reader is referred to the source code of the API for further intricacies.

*OpenSSL Engines*

The OpenSSL engines are what facilitates the integration of all other components of the design into OpenSSL. Recall from Chapter 1 that the OpenSSL high-level cryptographic API, known as the envelope (EVP) interface, has been re-organized to include a new API called the "Engine API" that supports alternative cryptography implementations and hardware acceleration. Third party vendors can request that their engines be included in the OpenSSL source tree, or the Engine code can be loaded dynamically at runtime as a shared library through a special built-in engine called "dynamic". If an engine for a cryptographic algorithm is loaded in a running OpenSSL instance, then whenever OpenSSL needs to perform this cryptographic operation, it will call the engines implementation of the algorithm rather than call its own.

Engines are compiled as a shared library, and dynamically loaded at runtime by OpenSSL when an engine is requested. Each engine must define several template functions that are required by the EVP interface to complete the respective cryptographic operations that otherwise would be performed in software. This chapter will use the SHA256 engine as an exemplar for how engines are designed, as the AES and RSA engines follow a similar series of steps. For more specific details on the RSA and SHA engines, please refer to the source code.

Before diving into writing an engine for a digest algorithm, it is worth demonstrating the minimal amount of code required for an engine to load into OpenSSL. The following code

based on a tutorial by Richard Levitte (45), shows an engine called “TestEngine”, whose only functionality is to be loaded and unloaded into a running OpenSSL instance.

```
1  #include <stdio.h>
2  #include <openssl/engine.h>
3
4  static const char *engine_id = "TestEngine";
5  static const char *engine_name = "A simple test engine";
6
7  static int engine_bind(ENGINE *e, const char *id)
8  {
9    int ret = 0;
10
11   if (!ENGINE_set_id(e, engine_id)) {
12     fprintf(stderr, "ENGINE_set_id failed\n");
13          return ret;
14   }
15   if (!ENGINE_set_name(e, engine_name)) {
16     printf("ENGINE_set_name failed\n");
17     return ret;
18   }
19
20   ret = 1;
21   return ret;
22 }
23
24 IMPLEMENT_DYNAMIC_BIND_FN(engine_bind)
25 IMPLEMENT_DYNAMIC_CHECK_FN()
```

The first thing to notice are the two macros on line 24 and 25. When an engine is first loaded, the OpenSSL library will perform certain checks to see that the engine is compatible with the current OpenSSL version; this is performed via the `IMPLEMENT_DYNAMIC_CHECK_FN()` macro. Next, the engine needs to “bind” with OpenSSL, which ties the engines functionality in with the running instance. A bind function is registered via the `IMPLEMENT_DYNAMIC_BIND_FN(*function)` macro. The bind function in TestEngine registers the engine ID and a string description of the engine with OpenSSL. Because this engine doesn’t do anything, nothing else needs to occur in the bind function. Later, the bind function will be where the engine should notify OpenSSL of its intended algorithm implementation. Assuming the above source code is in a file called test-engine.c, the TestEngine can be compiled as follows:

```
$  gcc -fPIC -o test-engine.o -c test-engine.c
$  gcc -shared -o test-engine.so -lcrypto test-engine.o
```

The first command compiles `test-engine.c` into the object file `test-engine.o` and ensures the code is position independent, meaning that the code can be executed from anywhere in memory regardless of its absolute address. This is a necessary step, since engines must be dynamically loaded shared objects, and therefore cannot be statically compiled. The second command links `test-engine.o` against the OpenSSL crypto library `libcrypto.so` to create a shared library object called `test-engine.so`. The engine can then be loaded on the command line using the following command:

```
$ openssl engine -t -c `pwd`/test-engine.so
 (/home/brett/tmp/test-engine/test-engine.so) A simple test engine
 Loaded: (TestEngine) A simple test engine
     [ available ]
```

This boilerplate code can be expanded to implement an engine for the SHA256 digest. From an engine point of view, there are three things that need to be done to implement a digest (46):

1. Create an OpenSSL digest method structure with pointers to the functions that will be OpenSSL's interface to the reference implementation.
2. Create the interface functions.
3. Create a digest selector function.

The OpenSSL digest method structure that must be implemented (item 1 above) is located in the OpenSSL source code at `include/internal/evp_int.h`

```
1 struct evp_md_st {
2     int type;
3     int pkey_type;
4     int md_size;
5     unsigned long flags;
6     int (*init) (EVP_MD_CTX *ctx);
7     int (*update) (EVP_MD_CTX *ctx, const void *data, size_t count);
```

```
 8      int (*final) (EVP_MD_CTX *ctx, unsigned char *md);
 9      int (*copy) (EVP_MD_CTX *to, const EVP_MD_CTX *from);
10      int (*cleanup) (EVP_MD_CTX *ctx);
11      int block_size;
12      int ctx_size;
13      int (*md_ctrl) (EVP_MD_CTX *ctx, int cmd, int p1, void *p2);
14  } /* EVP_MD */ ;

15  typedef struct evp_md_st EVP_MD;
```

The SHA256 engine must re-implement these functions through its own `EVP_MD` structure in order to adhere to the EVP API. The engine reimplementation for SHA256 is as follows:

```
 1  static EVP_MD sha256engine_sha256_method =
 2  {
 3      NID_sha256, // openSSL algorithm numerical ID
 4      NID_undef,  // pkey type
 5      DIGEST_SIZE_BYTES,          // message digest size (32 bytes)
 6      EVP_MD_FLAG_PKEY_METHOD_SIGNATURE, // default flags for digests
 7      sha256engine_sha256_init,     // digest initialization pointer
 8      sha256engine_sha256_update,   // digest update pointer
 9      sha256engine_sha256_final,    // digest final pointer
10      sha256engine_sha256_copy,     // copy function pointer
11      sha256engine_sha256_cleanup, // cleanup function pointer
12      NULL, // pointer to a function to sign data with a private key
13      NULL, // pointer to a funtion to verify signed with a public key
14      {NID_undef, NID_undef, 0,0,0}, // required pkey type
15      64, // standard SHA256 uses a block size of 512 Bits = 64 bytes
16      32, // context size (how big ctx->md_data must be)
17      NULL // pointer to md_ctrl control function
18  };
```

Lines 3-4 point the new `EVP_MD` `type` and `pkey_type` fields to the appropriate internal algorithm IDs defined in the `ossl_typ.h` header file. Line 5 defines the size of the message digest for SHA256. Lines 7-11 declare the interface function pointers for the SHA256 engine that will implement the actual cipher. It should be noted that the AES and RSA follow the same steps, but re-implement the `EVP_CIPHER` structure, rather than `EVP_MD`. The implemented functions are largely the same for both.

The first interface function to implement is the init function, which is called by OpenSSL when the engine is loaded. It receives a pointer to an `EVP_MD_CTX` structure from the

calling OpenSSL function. This structure holds state information about the digest, and is similar to the `sha256_ctx` structure detailed in the HLS algorithm in Chapter 3. The init function for the engine is as follows:

```
1  static int wssha256engine_sha256_init(EVP_MD_CTX *ctx)
2  {
3      ctx->update = &wssha256engine_sha256_update;
4      if (sha256_init() < 0)
5      {
6          fprintf(stderr,"SHA256 context could not be initialized\n");
7          return FAIL;
8      }
9      return SUCCESS;
10 }
```

The init function assigns the `EVP_MD_CTX` update function pointer and then calls the API initialization function from Chapter 3, which initializes the hardware to prepare for digest computation. The next interface function implemented is the core digest computation function:

```
1  static int sha256engine_sha256_update(EVP_MD_CTX *ctx,
                                         const void *data, size_t count)
2  {
3      unsigned char *digest = (unsigned char*)malloc(
                            sizeof(unsigned char) * DIGEST_SIZE_BYTES);
4      uint32_t digest_len = DIGEST_SIZE_BYTES;
5
6      // Call userspace API function for LKM
7      int status = sha256((uint8_t*)data, count,
                           (uint8_t*)digest, &digest_len);
8      ctx->md_data = digest; // copy data into context structure
9      if (status < 0)
10     {
11         fprintf(stderr,"ERROR: SHA256 algorithm failed\n");
12         return -1;
13     }
14     return SUCCESS;
15 }
```

This function calls the user space API digest function from Chapter 3, and copies the data back into the `EVP_MD_CTX` structure. All the computation happens in hardware, with the engine serving as a middleman between the OpenSSL instance and the kernel module. The

next function implemented is the final digest function, which simply copies the final data back into the `EVP_MD_CTX` buffer.

```
1 static int wssha256engine_sha256_final(EVP_MD_CTX *ctx,
                                         unsigned char *md)
2 {
3     memcpy(md, (unsigned char*)ctx->md_data,DIGEST_SIZE_BYTES);
4     return SUCCESS;
5 }
```

The final thing any digest engine must do is implement a digest selector function. The digest selector function is a function called by OpenSSL to determine whether the engine can support the requested digest algorithm. An engine can support multiple digest algorithms (sha256, sha1, md5, etc.) so this function is required to choose the appropriate implementation for the requested algorithm.

```
1 static int wssha256engine_digest_selector(ENGINE *e,
                    const EVP_MD **digest, const int **nids, int nid)
2 {
3     // if digest==null, return 0-terminated array of supported NIDs
4     if (!digest)
5     {
6         *nids = wssha256_digest_ids;
7         int retnids = sizeof(wssha256_digest_ids - 1) /
                      / sizeof(wssha256_digest_ids[0]);
8         return retnids;
9     }
10
11    // if digest is supported, select our implementation,
      // otherwise set to null and fail
12    if (nid == NID_sha256)
13    { // select our hardware digest implementation
14        *digest = &wssha256engine_sha256_method;
15        return SUCCESS;
16    }
17    else
18    {
19        *digest = NULL;
20        return FAIL;
21    }
22 }
```

OpenSSL calls this function in two ways, with the digest argument equal to `NULL`, or with a non-`NULL` digest argument. OpenSSL passes a `NULL` digest argument when it is attempting to determine which digests are supported by the engine. In this case, the engine

should set *nids to a zero-terminated array of supported NIDs and returns the number of available NIDs. If OpenSSL passes a non-NULL digest argument, then *digest is expected to be assigned the pointer to the EVP_MD structure corresponding to the NID given by nid. The call returns 1 if the requested NID is one supported by this engine, otherwise returns 0. It should be noted that for ciphers like the RSA and AES algorithms, a cipher selector function is implemented instead of a digest selector function; this function should work in exactly the same way.

Now that the engine supports SHA256, the following modifications to the bind function must be made

```
1  static int bind(ENGINE *e, const char *id)
2  {
3      int ret = FAIL;
4      if (!ENGINE_set_id(e, engine_id)) {
5          fprintf(stderr, "ENGINE_set_id failed\n");
6          return ret;
7      }
8      if (!ENGINE_set_name(e, engine_name)) {
9          fprintf(stderr,"ENGINE_set_name failed\n");
10         return ret;
11     }
12     if (!ENGINE_set_init_function(e, wssha256_init)) {
13         fprintf(stderr,"ENGINE_set_init_function failed\n");
14         return ret;
15     }
16     if (!ENGINE_set_digests(e, wssha256engine_digest_selector)) {
17         fprintf(stderr,"ENGINE_set_digests failed\n");
18         return ret;
19     }
20     ret = SUCCESS;
21     return ret;
21 }
```

The new additions to bind() point OpenSSL to the newly created init and digest selector functions

## The Yocto Project

This section will introduce the Yocto project, and detail how it was used to generate a customized Linux image for the Zynq that contains the kernel and application layer software described earlier in this chapter. The Yocto project hosts a suite of open source tools intended to help developers create their own custom Linux distributions for any hardware architecture and across multiple market segments (47). These tools are bundled together into a reference distribution called "Poky", which contains all the necessary infrastructure, tools, and metadata to automate the building and testing of a fully-customized embedded Linux distribution. Poky serves as a base system that can be modified and extended to quickly obtain a functional system.

### *OpenEmbedded and BitBake*

The core of the Yocto project is the OpenEmbedded (OE) build system, which features a powerful task execution tool called BitBake. BitBake is a generic task execution engine that allows shell and Python tasks to be run efficiently and in parallel while working within complex inter-task dependency constraints (48). OE comprises the set of metadata that describes all aspects of how the target image and associated software should be built. BitBake takes the task descriptions and metadata provided by OE as input, and generates a customized embedded Linux software stack. It is appropriate to view BitBake as a more complex and extensible version of GNU Make, designed from the ground up to cross-compile embedded Linux images and software applications.

OE organizes metadata into *layers* and *recipes*. A *recipe* is the most basic type of metadata, serving to instruct how BitBake should fetch, build, and install a particular portion of an

image. The relationship between a recipe and BitBake is roughly analogous to the relationship between a Makefile and the GNU Make program. Recipes provide BitBake with descriptive information about the package, versioning information, existing dependencies, where the source code resides and how to fetch it, how to configure and compile the source code, and where in the image to install the package or packages created. A *layer* is a bundle of metadata, which contains one or more recipes and other configuration files. Layers allow the user to isolate different types of image customizations from each other, enabling a more modularized system where individual portions can be added or

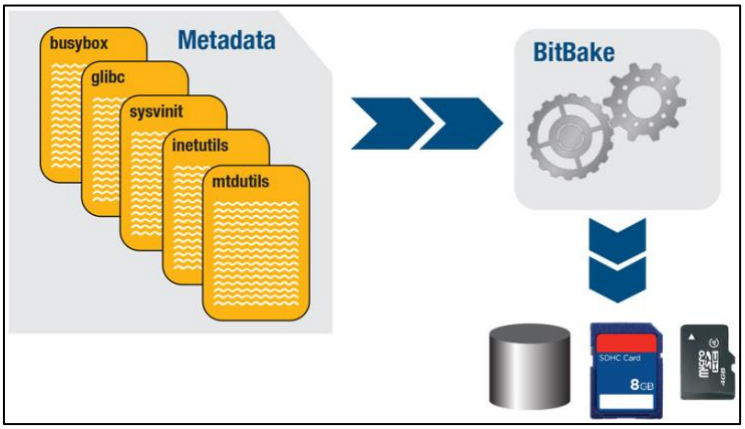


**Figure 14:** BitBake Recipe Building

subtracted from the final image without breaking dependency relationships. An illustration of BitBake building the recipes in a layer to make an image is shown in Figure 14.

*Terminology*

The terminology for the Yocto Project can be confusing. Therefore, each key component introduced in the previous section is outlined below for reference (48) (49):

- **Yocto** is an umbrella project, housing all the other tools mentioned in this section
- **Poky** is the Yocto reference distribution, built using the OpenEmbedded tools. It is an example image showing how all the tools bundled by Yocto should be used

- **OpenEmbedded (OE)** is the Yocto build system, comprised of metadata and the BitBake task execution engine
- **BitBake** is the generic task execution engine for OE, written in python. It parses recipes and configuration files to determine how best to assemble code from various sources into a useable image
- A **recipe** is a file with instructions for how BitBake should build something
- A **layer** is a bundle of OE metadata (recipes and configuration files) representing an independent "chunk" of the target image. Layers could represent a kernel, hardware support for a device, or an application stack
- An **image** is the binary output and associated root file system that is created by BitBake, intended to run on specific hardware or on an emulator

The Yocto project has excellent documentation and introductory material, therefore a step-by-step tutorial will not be provided in this thesis. Instead, the next section will provide an abbreviated "quick start" guide to building a generic image on an Ubuntu 16.04 host system. The motivated reader should refer to (47) (49) and (50) for further information.

*Obtaining Yocto and Building Poky for the Zedboard*

Before obtaining the Poky source, the host machine must be configured with the appropriate software packages that Yocto depends on. The following command installs the bare-minimal required packages to build an image that runs on QEMU (a virtualized hardware emulator) in graphical mode

```
$ sudo apt-get install gawk wget git-core diffstat unzip texinfo
   gcc-multilib build-essential chrpath socat cpio python python3
   python3-pip python3-pexpect xz-utils debianutils iputils-ping
   libsdl1.2-dev xterm
```

With the prerequisites installed, the next step is to obtain the source for poky using git, and check out the most recent project release by tag. To obtain the 2.4 release, execute the following commands:

```
$ git clone git://git.yoctoproject.org/poky
  Cloning into 'poky'...
  remote: Counting objects: 361782, done.
  remote: Compressing objects: 100% (87100/87100), done.
  remote: Total 361782 (delta 268619), reused 361439 (delta 268277)
  Receiving objects: 100%(361782/361782),131.94 MiB | 6.88 MiB/s, done.
  Resolving deltas: 100% (268619/268619), done.
  Checking connectivity... done.
$ git checkout tags/yocto-2.4 -b poky_2.4
```

Once the source is cloned onto the build machine (referred to as the "host"), check out the branch associated with the latest Yocto Project Release (currently "morty")

```
$ cd ~/poky
$ git checkout -b morty origin/morty
```

Git's checkout command checks out the current Yocto Project release into a local branch whose name matches the release (i.e. morty). The local branch tracks the upstream branch of the same name. Creating your own branch based on the released branch ensures you are using the latest files for that release (50).

The Poky reference project contains everything one needs to build an image using the Yocto tools. The first step in building the image is to set up the build environment. Yocto provides a script to do this, located at the top-level build directory.

```
$ source oe-init-build-env
```

Among other things, the script creates the Build Directory `build`, located in the Source Directory. After the script runs, your current working directory is set to the Build Directory.

Later, when the build completes, the Build Directory contains all the files created during the build (50).

General configuration information for the local build is contained in the file `build/conf/local.conf`. This file is where the target architecture is specified, along with other project specific items. Notice the line stating `MACHINE = "qemu-x86"`. This is the target architecture for which the image is compiled. This can be changed to any architecture with an OE board support package (BSP). For example, to target the Xilinx Zynq 7020, the layer (created by Xilinx) that contains BSPs for each of its development boards must be cloned.

```
$ git clone https://github.com/Xilinx/meta-xilinx.git
$ git checkout –b morty
```

Once the BSP is obtained (cloned into the top-level directory) via the previous command, the `MACHINE` parameter from the `local.conf` file can be set to any of the newly obtained Xilinx architectures

```
$ echo 'MACHINE = "zedboard-zynq7"' >> conf/local.conf
```

This is all that is needed to switch the target architecture of the entire build. Once the new target is in place, the image can be built by a direct invocation to `BitBake`

```
$ bitbake core-image-minimal
```

This command instructs BitBake to build the "core-image-minimal" recipe. The first build will take a long time (~45 minutes on a 2012 MacBook Pro with 16GB RAM and 2.8 GHz Intel Core i7). Successive builds will be much faster, since BitBake heavily employs caching via the shared state cache (51). When the build finishes, the important output files will be located at `build/tmp/deploy`. The files of interest are

- `boot.bin`: the first stage boot loader
- `u-boot.img`: the second-stage Linux boot loader
- `uImage`: the u-boot compatible Linux kernel image
- `uEnv.txt`: the command line arguments passed to the second stage bootloader
- `core-image-minimal-zedboard-zynq7.tar.gz`: the root filesystem

These files should be copied over to an appropriately partitioned SD card, as detailed in (52), (53), and (54). For more information on the Zynq boot process, including why these files are necessary, refer to chapter 6 of the Zynq technical reference manual (55).

*Customizing the Image to Support Hiding in Hardware*

The components of the design put forth in this thesis are integrated into the Yocto image via the `meta-cryptohw` layer. Recall from chapter 1 that the design is segmented into three "layers": the hardware, kernel, and application layers. These "layers" are logical groupings, and should not be confused with "Yocto/OpenEmbedded (OE) layers", which have a much stricter definition. In the remainder of this chapter, "layer" will refer to the

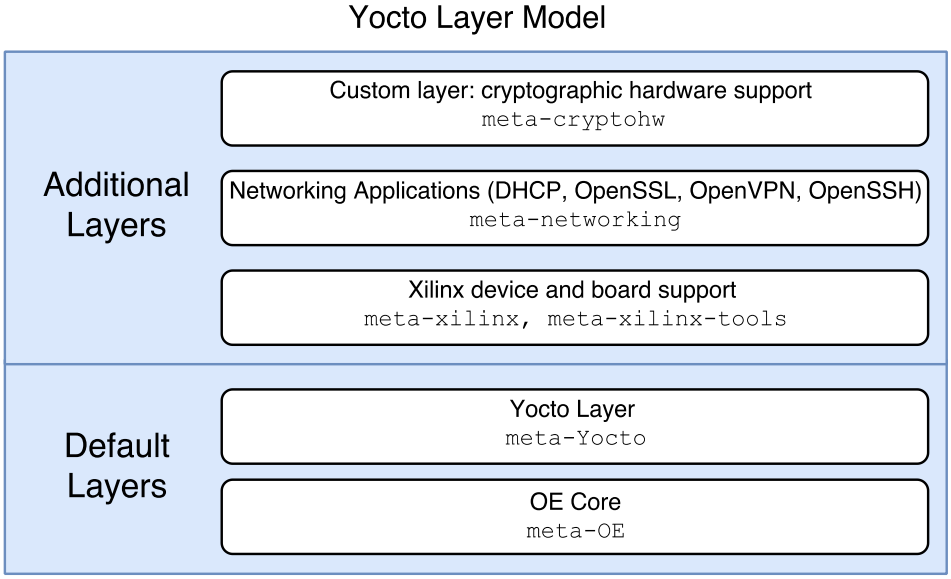


**Figure 15:** OE layer Model

logic layers of the design put forth in chapter 1, and "OE layer" will refer to the specific concept of a Yocto/OpenEmbedded software layer. The decision to wrap all layers of the design into a single OE layer was purely for simplicity and ease of distribution; it would be just as valid to place each design layer into its own OE layer, since the software stack for each cryptographic algorithm can stand on its own. The layering model for the Yocto image is shown in

.

The `meta-cryptohw` layer contains three directories, each containing the recipes for supporting one of the three hardware algorithms. The organization of the recipes within the layer is shown in Figure 16.

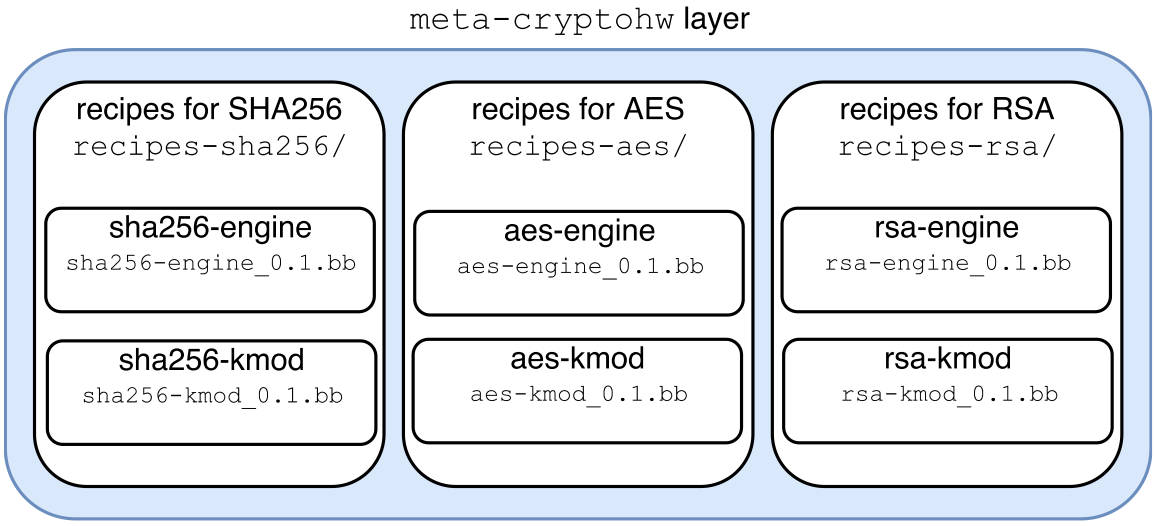


**Figure 16:** `meta-cryptohw` recipes

Recall that a *recipe* is simply a file that tells BitBake how to fetch, build, and install a portion of the target image. A recipe file, identified by its `*.bb` file extension, contains instructions written in a mix of Python and shell script that inform BitBake how to accomplish these tasks. As shown above in Figure 16, the recipes pertaining to each

hardware algorithm are grouped in a directory called `recipes-XX`, where `XX` is the algorithm in question. Each algorithm has two recipes: one supporting the application layer OpenSSL engine, and another which supports the kernel layer loadable kernel module. There is no recipe supporting each algorithm's hardware layer because, at the time of writing, Xilinx does not provide full support for executing synthesis runs using BitBake. Although scripts could be written to accomplish this task without any built-in Xilinx support, it would be a purely academic exercise and not necessary in the proposed system. Any time the underlying hardware is changed, the Zynq PL can be easily reprogrammed with a simple shell command; therefore, recipes supporting the hardware layer were not included into the OE layer for this design.

The two recipes supporting each hardware algorithm shall now be examined, using the SHA256 algorithm as an example. The recipes for AES and RSA are almost identical, and so will not be discussed. The recipe for integrating in the SHA256 kernel layer is shown below:

```
 1 SUMMARY = "Linux driver for SHA256 crypto hardware on the Zedboard"
 2 LICENSE = "MIT"
 3 LIC_FILES_CHKSUM =
"file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"
 4 # We use cmake as the local build system for the source code
 5 inherit module cmake
 6 # Here we provide the URL of the remote repository hosting the code
 7 SRC_URI = "git://github.com/username/my_repo.git;branch=master"
 8 SRCREV = "${AUTOREV}"
 9 # Establish the source directory in the build tree
10 S = "${WORKDIR}/git"
11
12 # Add the .so to the main package's files list
13 FILES_${PN} += " ${libdir} \
14                  ${bindir} \
15                  ${libdir}/*.so \
16                  ${libdir}/*.a \
17                  ${bindir}/sha256-kmodtest"
18
```

```
19 # Ensure that the DEV package doesn't grab them first
20 # commenting this out did not change anything
21 FILES_SOLIBSDEV = ""
22
23 EXTRA_OECMAKE = "-DKERNEL_SRC=${STAGING_KERNEL_DIR} "
24
25 # Inheritors of module.bbclass automatically name module packages
26 # with "kernel-module-" prefix required by the oe-core environment.
27 do_install() {
28 # install uapi libraries
29       install -d ${D}${libdir}
30       install -m 0755 ${B}/lib/*.so ${D}${libdir}
31       install -m 0755 ${B}/lib/*.a ${D}${libdir}
32 # install test binary
33       install -d ${D}${bindir}
34       install -m 0755 ${B}/test/sha256-kmodtest ${D}{bindir}
35 }
```

Lines 1-3 above provide information about the purpose of the recipe and licensing. Line 5 allows BitBake to use built-in support for CMake, as opposed to standard GNU Make, since CMake was chosen as the build system for the kernel layer and application layer source code. This was an important design decision, as the subcomponents of the design should be able to compile and run regardless of whether the Yocto tools are in use. Line 7 provides the URL of the remote repository for the source code. At the time of writing, the source code for all portions of the design is hosted on a private GitHub repository, however BitBake can pull from any web hosting services, as well as public and private version control servers (48). Lines 13-17 register the important files and directories that should be included in the final “package”, which is simply a grouping of the files to be distributed and/or installed on the target image as a result of the build process. Line 23 overrides the `KERNEL_SRC` CMake variable to point at the target kernel source (in the Yocto build tree) as opposed to the default value – the kernel source of the host computer.

Lines 23-34 represent the primary BitBake operation provided by this recipe: a custom `do_install()` task. As far as this thesis is concerned, BitBake performs three core tasks

while processing each recipe: `do_fetch()`, `do_compile()`, and `do_install()`. There are many other standard tasks that BitBake can perform, including code unpacking, patching, configuration, and packaging. However, these operations are not used in this thesis, and therefore will not be discussed. The recipe above only needs to provide a `do_install()` function because the default `do_fetch()` and `do_compile()` tasks are left to their default implementations – the default implementation of `do_fetch()` is all that is needed to pull from a typical GitHub remote repository, and the `do_compile()` task is handled by CMake because of the "inherit cmake" statement on line 5. The only non-default task is `do_install()`, which simply installs the build output to the desired location in the target root filesystem.

The second recipe for SHA256 integrates the application layer into the target image. The recipe file is shown below.

```
 1 SUMMARY = "sha256 engine support for openSSL"
 2 SECTION = "examples"
 3 LICENSE = "MIT"
 4 LIC_FILES_CHKSUM =
"file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"
 5 DEPENDS = "openssl"
 6
 7 SRC_URI = "git://github.com/username/myrepo.git;branch=master"
 8 SRCREV = "${AUTOREV}"
 9
10 # Add the .so to the main package's files list
11 FILES_${PN} += " ${libdir} \
12                  ${bindir} \
13                  ${libdir}/*.so \
14                  ${bindir}/sha256enginetest "
15
16 # Ensure that the DEV package doesn't grab them first
17 FILES_SOLIBSDEV = ""
18 # Establish the source directory in the build tree
19 S = "${WORKDIR}/git"
20
21 # Override important makefile variables for cross compilation
22 # within the Yocto build environment
23 PARALLEL_MAKE = ""
```

```
24 EXTRA_OEMAKE = "'CC=${CC}' \
25                 'LIB=-Llib -L=${libdir} ${LDFLAGS} -lcrypto -lsha'"
26
27 do_install() {
28 # install engine shared library
29       install -d ${D}${libdir}
30       install -m 0755 ${S}/bin/libsha256engine.so ${D}${libdir}
31 # install test binary
32       install -d ${D}${bindir}
33       install -m 0755 ${S}/bin/sha256enginetest ${D}{bindir}
34 }
```

Notice that both recipes are almost identical, with the only exception being that the application layer recipe does not contain support for CMake. The OpenSSL source code uses GNU Make as its build system, so to ensure simplicity and compatibility, the same tool was chosen to build the engine code.

When the BitBake command is run on any of the individual recipes (e.g. `bitbake sha256-kmod` or `bitbake sha256-engine`), the following operations are exectued:

1. source code is fetched from GitHub and extracted to the workspace in the poky/build/ directory
2. Depending on the recipe's build system, either the `make` or `cmake` command is executed in the workspace directory
3. The build outputs from the previous step are installed at the location in the target image specified in the recipe's `do_install()` step

# Ch. 5: Testing and Performance Analysis

This chapter analyzes the improvements to security and the performance impacts of hiding core OpenSSL cryptographic algorithms in hardware. Recall from Chapter 1 that Security, Performance, and Logic utilization were the chosen metrics for benchmarking the efficacy of the design, on both the hardware and systems level.

## Security

### *Inherent Security Improvement from Hiding in Hardware*

In Chapter 1 it was shown that most existing network security mechanisms rely on software-based techniques, due to their ease of implementation, ease of updating, and the ability to be dynamically deployed in anticipation or in response to threats (56). Because software-based mechanisms typically execute on the same processors they are attempting to protect, an attacker may achieve sufficient privilege on the processor to observe, alter, and defeat the security mechanism (10). Hardware security mechanisms do not suffer from this problem because hardware is essentially immutable: exotic tools and techniques, such as acid etching and X-ray reverse engineering, as well as physical access to the chip, are required to alter the behavior of hardware-based logic. Therefore, the migration of critical TLS algorithms and data structures into hardware serves as a *qualitative* measurement of security hardening and improved resilience – it increases attacker workload such that the timescale to breach the system makes the attack irrelevant, or the cost of attack is prohibitive when compared to the value of the potential outcome (56). Hiding in hardware prohibits application and kernel level visibility of sensitive data structures, even if attacker code is elevated to the highest processor privilege level (15).

*Attack Surface Reduction: OpenSSL Source Lines of Code*

Recall from Chapter 1 that recent studies have shown a directly proportional relationship between the number of source-lines-of-code (SLOC) in open-source software and the number of exploitable vulnerabilities, with an average of 0.61 programming defects for every thousand lines of code (8). This section enumerates the improvement in security based on the proportion of OpenSSL source code hidden in hardware.

It is difficult to quantify exactly how much code can be removed from the OpenSSL source by hiding key algorithms in hardware, since the dynamic engine API was designed to allow alternative implementations of its cryptographic algorithms without requiring any modification to the source code. Additionally, there are numerous references to the raw cryptographic data structures and functions throughout the OpenSSL source code that exist outside the scope of the EVP API, making it even harder to determine whether said code should be removed without altering other functionality. Therefore, the SLOC reduction numbers in this section represent a *best-effort* attempt at stripping the default cryptographic algorithm implementations from the source tree, performed with simple text parsing tools and linker output to identify the relevant code sections for removal. It is important to note that these results represent a lower bound for how much code can be removed; sections of relevant code were inevitably missed, and a restructuring of the source code to remove the overhead required to support these algorithms in software would clearly reduce SLOC even further. SLOC are counted using the open-source `cloc` utility, written by Al Danial (57).

It was found that approximately **15,462** lines of code were removed from the OpenSSL source code by hiding the SHA256, AES, and RSA algorithms in hardware. This constitutes an absolute minimum SLOC reduction of approximately **6%** for the entire OpenSSL project. Considering the average frequency of programming defects per SLOC found in (8), a 6% SLOC reduction in OpenSSL translates to the elimination of approximately 156 potential vulnerabilities. Because SHA256, AES, and RSA comprise the minimum set of cryptographic algorithms necessary to establish a valid TLS session, it is also worth analyzing the SLOC reduction yielded by a system that *only* supports these three core algorithms in hardware. By removing software support for every software algorithm that is compiled into the standard OpenSSL distribution, an overall SLOC reduction of approximately **40%** can be achieved. The resulting SLOC reductions are tabulated below in Figure 17.

| **Algorithms Hidden in Hardware** | **None** | **SHA256, AES, RSA** | **All** |
|---|---|---|---|
| **OpenSSL SLOC** | 271,282 | 255,820 | 109,801 |
| **Approx. SLOC Reduction (%)** | 0% | 6% | 40% |

**Figure 17:** SLOC Reduction Results

**Hardware Performance**

This section assesses the hardware cost and performance impact of hiding the cryptographic algorithms in FPGA logic, using the standard metrics of logic utilization, latency, and throughput, obtained from the Vivado HLS synthesis report. Results only pertain to the pure hardware performance; the overhead associated with Linux and OpenSSL is discussed in the next section. It should also be noted that once the hardware cores are converted to function as bus-masters, the performance of each algorithm will dramatically improve due to the increased bus-transfer bandwidth. Recall from chapter 1

that AXI slaves are limited to single word read/write transactions across the memory bus. Once converted to act as bus masters, the algorithms will be able to use high-performance 256-word (32-bit or 64-bit) burst transactions.

*SHA256*

The SHA256 algorithm, when implemented as an AXI Slave (single-word reads/writes across the AXI bus) demonstrated a minimum and maximum estimated latency of 494 and 113,862 clock cycles, respectively. At a clock period of 8 ns, this results in a maximum hardware latency of $113{,}962\ cycles * \frac{8\ ns}{cycle} = 0.911\ miliseconds$ to compute a 256-bit digest. Given that the initiation interval is one clock cycle more than the maximum hardware latency, and each interval processes a 512 bit (64 byte) input, the worst-case theoretical bandwidth of the pure hardware implementation is computed to be $\frac{64\ bytes}{0.911\ ms} = \mathbf{70.26\ kB/s}$, and a best-case theoretical bandwidth of **16.78 MB/s.** The estimated utilization of the SHA256 hardware block is modest, with **no more than 12% utilization** of any type of FPGA resource. It should be noted that post-synthesis utilization is only an estimate, as the logic utilization for a given hardware block is highly dependent upon other hardware cores present in the design, and therefore cannot be determined until

**Performance Estimates**

**Timing (ns)**

**Summary**

| Clock | Target | Estimated | Uncertainty |
|---|---|---|---|
| ap_clk | 12.00 | 9.07 | 1.50 |

**Latency (clock cycles)**

**Summary**

| Latency | | Interval | | |
|---|---|---|---|---|
| min | max | min | max | Type |
| 494 | 113962 | 495 | 113963 | none |

**Utilization Estimates**

**Summary**

| Name | BRAM_18K | DSP48E | FF | LUT |
|---|---|---|---|---|
| DSP | - | - | - | - |
| Expression | - | - | 251 | 188 |
| FIFO | - | - | - | - |
| Instance | 10 | - | 7367 | 5290 |
| Memory | 2 | - | 0 | 0 |
| Multiplexer | - | - | - | 1003 |
| Register | - | - | 1242 | - |
| Total | 12 | 0 | 8860 | 6481 |
| Available | 280 | 220 | 106400 | 53200 |
| Utilization (%) | 4 | 0 | 8 | 12 |

**Figure 18:** SHA256 Hardware Performance

implementation. The post-synthesis performance and utilization results provided by Vivado HLS are shown in Figure 18.

*AES*

The AES algorithm, implemented as an AXI slave, demonstrated a minimum and maximum estimated latency of 22 and 2,544 clock cycles, respectively. At a clock period of 8 ns, this results in a maximum hardware latency of $2{,}544\ cycles * \frac{8\ ns}{cycle} = 20.35\ microseconds$. Given that AES processes 16 bytes of input data per initiation interval, the worst-case theoretical bandwidth of the pure hardware implementation is computed to be $\frac{16\ bytes}{20.35\ \mu s} = \mathbf{786.24\ kB/s}$, and a best-case theoretical bandwidth of **90.91 MB/s.** Similar to SHA256, the estimated hardware utilization of AES is modest, with **no more than 11% utilization** for any type of FPGA resource. The post-synthesis performance and utilization results provided by Vivado HLS are shown in Figure 19.

**Performance Estimates**

**Timing (ns)**

**Summary**

| Clock | Target | Estimated | Uncertainty |
|---|---|---|---|
| ap_clk | 12.00 | 8.53 | 1.50 |

**Latency (clock cycles)**

**Summary**

| Latency | | Interval | | |
|---|---|---|---|---|
| min | max | min | max | Type |
| 1 | 2543 | 2 | 2544 | none |

**Utilization Estimates**

**Summary**

| Name | BRAM_18K | DSP48E | FF | LUT |
|---|---|---|---|---|
| DSP | - | - | - | - |
| Expression | - | - | 340 | 230 |
| FIFO | - | - | - | - |
| Instance | 12 | - | 2193 | 4741 |
| Memory | 3 | - | 80 | 14 |
| Multiplexer | - | - | - | 957 |
| Register | - | - | 268 | - |
| Total | 15 | 0 | 2881 | 5942 |
| Available | 280 | 220 | 106400 | 53200 |
| Utilization (%) | 5 | 0 | 2 | 11 |

**Figure 19:** AES Hardware Performance

*RSA*

The RSA algorithm, implemented as an AXI slave, demonstrated a minimum and maximum estimated latency of 6,323,361 and 12,618,913 clock cycles, respectively. At a clock period of 12ns, this results in a maximum hardware latency of $12{,}618{,}913\ \ cycles * \frac{12\ ns}{cycle} = 151.43\ \ miliseconds$. Given that the RSA 1024 algorithm processes 128 bytes per initiation interval, the worst-case theoretical bandwidth of the pure hardware implementation is computed to be $\frac{128\ bytes}{151.43\ ms} = \mathbf{0.84\ kB/s}$, and a best-case theoretical bandwidth of **1.69 kB/s**. Similar to SHA and AES, the RSA algorithm used a modest number of BRAM and LUT resources. However, unlike SHA and AES, the RSA algorithm

**Performance Estimates**

**Timing (ns)**

**Summary**

| Clock | Target | Estimated | Uncertainty |
|---|---|---|---|
| ap_clk | 12.00 | 11.10 | 1.50 |

**Latency (clock cycles)**

**Summary**

| Latency | | Interval | | |
|---|---|---|---|---|
| min | max | min | max | Type |
| 6323361 | 12618913 | 6323362 | 12618914 | none |

**Utilization Estimates**

**Summary**

| Name | BRAM_18K | DSP48E | FF | LUT |
|---|---|---|---|---|
| DSP | - | - | - | - |
| Expression | - | - | 92 | 73 |
| FIFO | - | - | - | - |
| Instance | 8 | - | 43840 | 7540 |
| Memory | - | - | - | - |
| Multiplexer | - | - | - | 270 |
| Register | - | - | 13170 | - |
| Total | 8 | 0 | 57102 | 7883 |
| Available | 280 | 220 | 106400 | 53200 |
| Utilization (%) | 2 | 0 | 53 | 14 |

**Figure 20:** RSA 1024 Hardware Performance

was estimated to utilize a significant portion of the available flip-flop (FF) resources. The highly sequential and iterative nature of the modular exponentiation algorithm prevents efficient resource sharing, causing the RSA hardware core to **require approximately 53% of the available FFs**. The post-synthesis performance and utilization results provided by Vivado HLS are shown in Figure 20.

*Bare-Metal Performance Summary*

**Bare-metal performance for all three algorithms implemented as AXI Slaves is summarized below in**

Figure 21.

| **Algorithm** | SHA | AES | RSA |
|---|---|---|---|
| **Max Latency (Clock cycles)** | 113962 | 2544 | 12618193 |
| **Initiation Interval (Clock cycles)** | 113963 | 2544 | 12618194 |
| **Min Clk Period (ns)** | 8 | 8 | 12 |
| **Max Init Bandwidth (inits/sec)** | 1052 | 47169 | 6 |

**Figure 21**: Summary of Hardware Performance

It should again be noted that these performance results represent an absolute lower bound on data bandwidth, given that AXI slaves are limited to single-word read/write memory transactions. Converting the hardware cores to function as AXI masters will dramatically increase the bandwidth and reduce the initiation interval of each algorithm.

Figure 22 shows an example of the memory access bottleneck incurred from the AXI slave interface, obtained from a Vivado HLS cycle analysis of the RSA hardware algorithm. Control steps performing memory accesses are enclosed within the red box annotations. The green blocks represent the actual computation of the algorithm, not involving AXI transactions. Note that the first 27 control steps (sequences of clock cycles) consist entirely of sequential reads from memory; this comprises the necessary step of loading the input data buffer with the initial value from memory. Similarly, from control step 55 and onwards,

the utilized clock cycles are entirely writes back to memory; this is the necessary step of copying the resultant data back into PS memory space over the AXI bus. The analysis tool cannot calculate exactly how many clock cycles these operations take, since they occur within in a loop that is escaped via a conditional statement; This value can be computed however, based on the known size of the input and output data. Three 1024-bit data buffers must be sequentially loaded before the modular exponentiation can be performed, and one 1024-bit data buffer must be written back to PS memory after the computation is complete. This equates to 128 distinct AXI transactions for every invocation of the algorithm, where each transaction incurs 100% of the overhead for AXI addressing. If this hardware core

used an AXI master interface, it would only require a single AXI burst transaction to load the input data buffer, and a single burst transaction to return the output buffer.

### System-level Performance

Performance of the software components on a system level was measured using OpenSSL's built-in performance benchmarking tool. This reflects the overarching performance of the entire system once integrated with OpenSSL, including any overhead incurred from the kernel and application layer software.

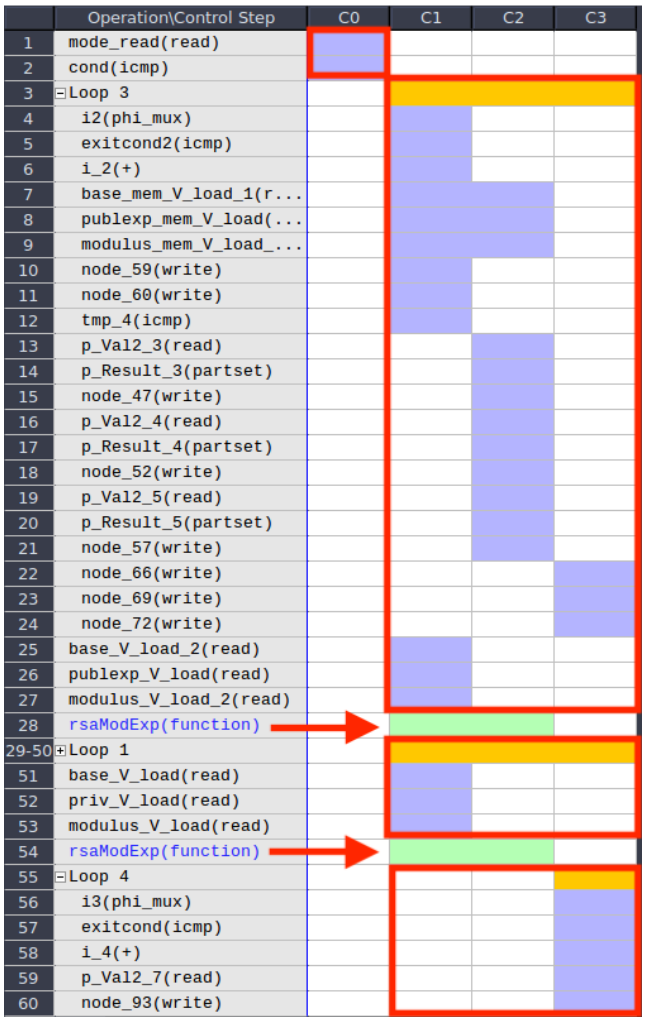


**Figure 22:** AXI Slave Memory Access Bottleneck

OpenSSL's built in performance monitoring tool, `openssl speed`, measures the average execution time required by a given cryptographic algorithm when repetitively invoked on a variety of input data sizes. Time is measured using either *elapsed time*, i.e. the "wall clock" time required for the algorithm to run to completion, or *CPU time*, the amount of time the time actually spent by the CPU executing relevant code. Elapsed time was chosen as the desired metric, since this most directly translates to the latency experienced by the end user when using hardware-assisted algorithms in a real world scenario. It should be readily apparent that a hardware assisted OpenSSL stack is dramatically more performant from the point of view of CPU time, since all cryptographic operations are performed in hardware, with no additional load imposed on the host processors.

Performance data for the standard OpenSSL software algorithms was obtained using the following command line invocation:

```
$  openssl speed -evp <algorithm> -elapsed
```

This command performs a speed test on the EVP implementation of the specified algorithm, using elapsed time as a metric. Results of the speed test are written to STDOUT. The same speed test was then performed on the hardware implementations of the same EVP algorithms, obtained using the following command line invocation:

```
$  openssl speed -evp <algorithm> -elapsed -engine /path/to/engine.so
```

*Analysis*

The performance results for the AXI slave implementations of SHA and AES are reported in Figure 23, averaging the results across 10 runs. Results were compared against a manual

timing test, using the Linux timer functionality from `<time.h>`, and were found to be consistent. No system-level timing data was obtained for the RSA due to the aforementioned memory access bug that was only present under Linux.

| **Algorithm** | **SHA** | | **AES** | |
|---|---|---|---|---|
| **Implementation** | **Software** | **Hardware** | **Software** | **Hardware** |
| **Avg. # computations/s** | 202,481.67 | 7,622.29 | 70,281.95 | 2,446.84 |
| **Avg. throughput (kB/s)** | 12,958.83 | 487.8 | 17,994.739 | 626.39 |

**Figure 23:** SHA256 and AES Performance Comparison

### Additional Benefits of HLS

Hardware Description Languages also introduce new lines of source code, however does not provide any data regarding the security impact of hardware code size (8). Although there may always be a potential correlation, from the perspective of attack surface reduction the argument is largely irrelevant because hardware code does not generate binary images observable by the system processor except where intentionally exposed (e.g. through a memory interface presented to the processing system by an FPGA-based resource) (15). However, even if there is no proven impact on security, the reduction in code size gained in using HLS over a traditional HDL provides a nontrivial advantage to the hardware designer and reflects the motivation behind exploring HLS in this thesis. The increasing adoption of HLS tools demonstrates that the significant reduction in code complexity provided by HLS tends to outweigh the slight reduction in efficiency of the generated HDL for all but the most performance critical high-speed applications.

## Ch. 6: Future Work and Conclusions

This chapter summarizes the contributions put forth in this thesis, proposes directions for future work, and makes concluding remarks on the results presented in earlier chapters.

**Summary**

This thesis explored a novel approach to improving the security of OpenSSL by hiding encryption algorithms and secret keys within the perimeter of a single heterogeneous System-on-Chip (SoC) device: the Xilinx Zynq 7020. The proposed design greatly improves the security of OpenSSL by providing secure key storage and hiding custom cryptographic algorithms hidden in tightly-coupled FPGA hardware, establishing a *single-chip, hardware base-of-trust* that exposes no off-chip interconnects to reverse-engineering. The approach heavily employs the recently introduced "engine API", which allows OpenSSL to utilize alternative implementations of the default cryptographic algorithms specified at runtime. All the custom cryptographic hardware was developed using High-Level Synthesis, yielding a significant reduction in code complexity, development time, and maintenance costs, when compared to traditional HDL design. A customized minimal Linux operating system was created for the Zynq, with OpenSSL and software support for the custom cryptographic hardware preinstalled in the root filesystem. The operating system image was customized using the Yocto project tool suite, enabling it to be reconfigured for any target architecture without requiring any modification to the source code. The kernel and application layers of the design are fully self-contained inside an OpenEmbedded/Yocto layer, and can be distributed and integrated into any Yocto-based image regardless of the architecture or local build system.

## Future Work

*Hardware Layer*

Although implementing the cryptographic algorithms in hardware yields a dramatic improvement to system security due to the increase in attacker workload, there remain several optimizations to the hardware algorithms that would further harden the system against advanced exploitation mechanisms. Most notably, both the AES and RSA cipher algorithms have been shown to be vulnerable to a variety of *side-channel attacks* – an attack based on information gained from the physical implementation of a cryptosystem, rather than brute force or a theoretical weakness in the cryptographic algorithm itself (58). Side-channel attacks include exploiting information leaked by a cryptographic device's execution time (59), power consumption (60), electromagnetic radiation (61), and even sound (62), to recover a portion of the secret information. Informed by recent research, proposed future work to eliminate these side channels would include implementing constant-time version of the algorithms (63), performing dummy encryption on zero data (64), eliminating conditional branching and secret intermediates (60), adding dummy registers and additional logic to "randomize" power consumption (65), and many other well-documented techniques.

As mentioned in Chapter 1, a massive security increase could also be obtained by migrating the entire network stack into FPGA fabric. In the current design, only the cryptographic algorithms are hidden in hardware; valuable side channel information, network addresses, and unsecured traffic can still be intercepted by a third party with elevated processor privileges. This vulnerability would be eliminated with a hardware-based network stack,

and have the additional benefit of reducing the CPU overhead involved with maintaining a TLS session or running a VPN client. An excellent first step would be to move the Ethernet MAC into FPGA logic, potentially using a drop-in IP core or an HLS design, and then sequentially move additional components higher up in the OSI model into hardware; the end goal being the implementation of a full TLS-enabled network stack entirely hidden in the FPGA.

Finally, the largest area for future work in the hardware layer is the conversion of the hardware algorithms from AXI-lite slaves to AXI masters. Given the substantial complexity of the system developed in this thesis, it was decided to implement the hardware algorithms as slaves on the AXI bus due to ease of integration with the OpenSSL software stack. However, this design decision comes at a heavy performance penalty, with the bandwidth of the hardware severely limited by AXI4-lite's single word memory transactions. The AXI4-lite implementation requires two cycles per word read, resulting in a 50% transaction overhead. The full AXI4 implementation can read 256 sequential words in 256+1 cycles, resulting in less than 0.5% overhead (56). With less than 0.5% overhead, a 64-bit wide bus operating at 120 MHz would yield a theoretical burst-based throughput of approximately 7 Gbps per channel. With 50% overhead for read access, and a maximum of 32-bit transfers, the AXI-lite implementation yields a memory bandwidth of closer to 3 Gbps. For optimal hardware performance, the burst memory reads are clearly preferred. Thanks to the high level of abstraction provided by HLS, converting the algorithms to function as bus-masters is a trivial exercise at the hardware layer. Unfortunately, this requires a more complicated software interface at the kernel layer that was deemed beyond

the scope of this thesis. It is therefore left as proposed future work, and should be an immediate next step in the development of the system designed in this paper.

*Kernel Layer*

The kernel layer is immune from attackers with user-level presence on the system since it operates in kernel memory space, which requires elevated user privileges. However, should an attacker obtain privilege escalation, the entire kernel could be compromised. In this scenario, secret keys are still unable to be recovered from the hardware modules by design, but significant damage to the system could still occur (as it would on any other system where an attacker obtains privilege escalation). Therefore, future work should include the implementation of a hardware-based code monitor hidden in FPGA logic, such as that developed by Dahlstrom and Taylor in (56), which can immediately detect kernel code modifications and revert the active kernel to a known state. Because it is completely hidden in hardware, an attacker with full root-level privileges would still be unable to disable the monitor and evade detection after modifying process code (56) (15).

Additionally, future work for the kernel layer should include the development of kernel drivers supporting the AXI-master version of the hardware algorithms. This is a critical next step in the overall hardening of the system, and will dramatically improve system performance and security.

*Application Layer*

Proposed future work in the application layer consists of eliminating the need for OpenSSL to maintain a TLS session. As more cryptographic and networking components are migrated into hardware, there is no longer be a need for sensitive operations (such as TLS

management) to occur on the processors. All networking and cryptographic operations should eventually be handled entirely in hardware, and configured through a secure out-of-band channel.

## References

1. Dierks, T. and Rescorla, E. RFC5246: The Transport Layer Security (TLS) Protocol, Version 1.2. [Online] August 2008. https://tools.ietf.org/html/rfc5246.

2. Netcraft. 2014 Annual Web Server Survey. [Online] April 2014. http://news.netcraft.com/archives/2014/04/02/april-2014-web-server-survey.html .

3. OpenSSL . *Heardbleed Exploit Detail.* [Online] https://www.heartbleed.com.

4. Long, Michael C. Attack And Defend: Linux Privilege Escalation Techniques of 2016. . *SANS Institute InfoSec Reading Room .* [Online] 2017. https://www.sans.org/reading-room/whitepapers/testing/attack-defend-linux-privilege-escalation-techniques-2016-37562.

5. Xilinx . Field Programmable Gate Array (FPGA). *Xilinx .* [Online] https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html.

6. Grand View Research. Field Programmable Gate Array (FPGA) Market Analysis By Technology (SRAM, EEPROM, Antifuse, Flash), By Application (Consumer Electronics, Automotive, Industrial, Data Processing, Military & Aerospace, Telecom), And Segment Forecasts, 2014 - 2024. [Online] December 2016. http://www.grandviewresearch.com/industry-analysis/fpga-market.

7. Intel . Intel Completes Acquisition of Altera. *Intel Newsroom .* [Online] December 28, 2015. https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera/.

8. *Reliability Issues in Open Source Software.* Tiwari, V and Pandi, R.K. 2011, International Journal of Computer Applications, Vol. 34.

9. Synopsis Inc. . Coverity Scan Open Source Report 2014. [Online] 2015. http://go.coverity.com/rs/157-LQW-289/images/2014-Coverity-Scan-Report.pdf.

10. *Migrating an OS Scheduler into Tightly Coupled FPGA Logic to Increase Attacker Workload.* Dahlstrom, Jason and Taylor, Stephen. s.l. : AFCEA, IEEE, 2013. MILCOM.

11. Ramirez, Dr. Sergio. High Level Synthesis: Understanding Latency versus Throughput . *Cadence System Design and Verification Blogs.* [Online] Cadence Design Systems Inc. https://community.cadence.com/cadence_blogs_8/b/sd/archive/2010/09/13/understanding-latency-vs-throughput.

12. Xilinx . UG474: 7-Series FPGAs CLB User Guide. [Online] https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf.

13. Digilent. ZYBO FPGA Board Reference Manual. [Online] https://reference.digilentinc.com/reference/programmable-logic/zybo/reference-manual.

14. Xilinx. ds190: Zynq-7000 All Programmable SoC Data Sheet Overview. [Online] https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.

15. Dahlstrom, Jason. *Hiding in Hardware: Increasing System Security using Tightly-Coupled FPGA Logic.* Thayer School of Engineering, Dartmouth College. 2015. Ph.D. Thesis.

16. NIST Computer Security Reference Center. Hash Functions. [Online] January 1, 2017. https://csrc.nist.gov/projects/hash-functions/sha-3-project.

17. NIST. FIPS PUB 180-1: Secure Hash Standard. [Online] August 2015. http://doi.org/10.6028/NIST.FIPS.180-4.

18. CRIPTOGRAFIA MAII - FIB. The cryptographic hash function SHA-256. [Online] https://www.researchgate.net/file.PostFileLoader.html?id=534b393ad3df3e04508b45ad&assetKey=AS%3A273514844622849%401442222429260.

19. Conte, Brad. Basic Implementations of Standard Cryptograpy Algorithms. [Online] 2014. https://github.com/B-Con/crypto-algorithms.

20. NIST. FIPS PUB 197: Announcing the ADVANCED ENCRYPTION STANDARD (AES). [Online] November 26, 2001. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

21. Paar, Christof and Pelzl, Jan. *Understanding Cryptography, A Textbook for Students and Practitioners.* s.l. : Springer, 2010. 978-3-642-04101-3.

22. Daemen, Joan and Rijmen, Vincent. *The Design of Rijndael: AES- The Advanced Encryption Standard.* s.l. : Springer, 2002. 3540425802.

23. Schneier, Bruce, et al. The Twofish Team's Final Comments on AES Selection. [Online] May 2000. http://www.schneier.com/paper-twofish-final.pdf.

24. Mukhopadhyay, Sourav. The Advanced Encryption Standard (AES). *Indian Institute of Technology Kharagpur, MA60031: Cryptography and Network Security.* [Online] http://www.facweb.iitkgp.ernet.in/~sourav/AES.pdf.

25. Cobb, Michael, Hazan, Fred and Rundatz, Frank. RSA algorithm (Rivest-Shamir-Adleman). *SearchSecurity.* [Online] November 2014. http://searchsecurity.techtarget.com/definition/RSA.

26. *A Method of Obtaining Digital Signartures and Public-Key Cryptosystems.* Rivest, Ron, Shamir, Adi and Adleman, Leonard. February 1978, Communications of the ACM. 10.1145/359340.359342.

27. NIST. Special Publication 800-57 Part 3: Recommendation for Key Management. [Online] 1, January 2015. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57Pt3r1.pdf.

28. —. FIPS PUB 186-4: Digital Signature Standard (DSS). [Online] July 2013. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.

29. Kaliski, Burt. The Mathematics of the RSA Public-Key Cryptosystem. [Online] 2006.

30. Koç, Ç.K. *RSA Hardware Implemenation.* RSA Laboratories. 1995.

31. Wöckinger, Thomas. *High-Speed RSA Implementation for FPGA Platforms.* Institute for Applied Information Processing and Communications , Graz University of Technology. 2015. Master's Thesis.

32. *High-Radix Montgomery Modular Exponentiation on Reconfigurable Hardware.* Blum, Thomas and Paar, Christof. 5, May 2001, IEEE Transactions on Computers, Vol. 50.

33. Koç, Ç.K. *High-Speed RSA Implementation.* RSA Laboratories. 1994. Technical Report TR201.

34. *An Improved Unified Scalable Radix-2 Montgomery Multiplier.* Harris, David, et al. s.l. : IEEE, 2005. Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH'05). 1063-6889/05.

35. Hachez, G and Quisquater, J. Montgomery Exponentiation With No Final Subtraction: Improved Results. [ed.] C.K. Koç and C. Paar. *Lecture Notes in Computer Science.* 2000, Vol. 1965, pp. 293-301.

36. *An Optimized Hardware Architecture for the Montgomery Multiplication Algorithm.* Huang, M., et al. [ed.] R Cramer. Berlin : Springer, 2008. Lecture Notes on Computer Science, Public Key Cryptography – PKC 2008. Vol. 4939.

37. Loadable Kernel Modules. *Arch Linux Wiki.* [Online] https://wiki.archlinux.org/index.php/kernel_modules.

38. Bhattacharjee, Abhishek and Lustig, Daniel. *Architectural and Operating System Support for Virtual Memory.* s.l. : Morgan & Claypool Publishers, 2017. ISBN 9781627056021.

39. Salzman, Peter Jay, Burian, Michael and Pomerantz, Ori. The Linux Kernel Module Programming Guide . *The Linux Documentation Project.* [Online] 2.6.4, May 18, 2007. http://www.tldp.org/LDP/lkmpg/2.6/html/index.html.

40. Molloy, Derek. Writing A Linux Kernel Module Part 1: Introduction . *Linux Kernel Programming.* [Online] April 14, 2015. http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction/.

41. Hanrigou, Phillippe. In UNIX Everything is a File. *PH7.* [Online] Sept 2007. https://web.archive.org/web/20150110114929/http://ph7spot.com:80/musings/in-unix-everything-is-a-file.

42. Corbet, Jonathan, Rubini, Alessandro and Kroah-Hartman, Greg. *Linux Device Drivers.* [ed.] Andy Oram. Third Edition. Sebastopol : O'Reilly Media, Inc., 2005. https://lwn.net/Kernel/LDD3/.

43. Wikipedia. Ioctl. *Wikipedia.* [Online]
https://en.wikipedia.org/wiki/Ioctl?oldformat=true.
44. Xilinx Community Forums. *Xilinx.com.* [Online]
https://forums.xilinx.com/t5/Vivado-High-Level-Synthesis-HLS/HLS-axi-slave-behaves-incorrectly-under-linux-but-works-in-bare/m-p/790044#M9976.
45. Levitte, Richard. Engine Building Lesson 1: A Minimum Useless Engine. *OpenSSL Blog.* [Online] October 2015.
https://www.openssl.org/blog/blog/2015/10/08/engine-building-lesson-1-a-minimum-useless-engine/.
46. —. Engine Building Lesson 2: An Example MD5 Engine. *OpenSSL Blog.* [Online] November 2015. https://www.openssl.org/blog/blog/2015/11/23/engine-building-lesson-2-an-example-md5-engine/.
47. Yocto Overview. *Yocto Project.* [Online] Linux Foundation , 2012.
https://www.yoctoproject.org/question/what-yocto-project.
48. BitBake Community. BitBake User Manual. *The Yocto Project.* [Online] 2017.
http://www.yoctoproject.org/docs/2.4/bitbake-user-manual/bitbake-user-manual.html.
49. Yocto Project Reference Manual. *Yocto Project.* [Online] 2017.
http://www.yoctoproject.org/docs/2.4/ref-manual/ref-manual.html.
50. Yocto Project Quick Start. *Yocto Project.* [Online] 2017.
http://www.yoctoproject.org/docs/current/yocto-project-qs/yocto-project-qs.html.

51. Enable sstate cache. *Yocto Wiki.* [Online] 2017. https://wiki.yoctoproject.org/wiki/Enable_sstate_cache.

52. Building Linux For The Zynq ZC702 Eval Kit Using Yocto . *mbedded.ninja.* [Online] September 1, 2017. http://blog.mbedded.ninja/programming/embedded-linux/zynq/building-linux-for-the-zynq-zc702-eval-kit-using-yocto.

53. Xilinx. Prepare Boot Medium. *Xilinx Wiki.* [Online] October 2017. http://www.wiki.xilinx.com/Prepare+Boot+Medium.

54. Briggs, Tom. Building a Linux custom system for the ZedBoard with Yocto. *YouTube.* [Online] March 2017. https://youtu.be/XPnmB-THjiY.

55. Xilinx. UG585: Zynq-7000 All Programmable SoC Technical Reference Manual. [Online] v1.12, October 20, 2017. https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

56. *Hardware-Based Code Monitors on Hybrid Processor-FPGA System-on-Chip Architectures.* Dahlstrom, Jason and Taylor, Stephen. s.l. : AFCEA, IEEE, 2015. MILCOM.

57. Danial, A. CLOC - Count Lines of Code. [Online] 2012. https://github.com/AlDanial/cloc.

58. Wikipedia. Side-channel attack. *Wikipedia.* [Online] https://www.wikiwand.com/en/Side-channel_attack.

59. *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS and Other Systems.* Kocher, P. Santa Barbara, CA : s.n., 1996. International Cryptology Conference (CRYPTO). Vol. 1109, pp. 104-113.

60. *Differential Power Analysis.* Kocher, P., Jaffe, B. Jun. Santa Barbara, CA : s.n., 1999. International Cryptology Conference (CRYPTO). Vol. 1666, pp. 398-412.

61. *The EM Side-Channel(s).* Agrawal, D., et al. Redwood City, CA : LNCS, 2002. CHES 2002. Vol. 2523, pp. 29-45.

62. *RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis.* Genkin, D., Shamir, A and Tromer, E. s.l. : IACR, 2013.

63. *Quantifying Timing-Based Information Flow in Cryptographic Hardware.* Mao, M., et al. 2015. IEEE/ACM International Conference on Computer-Aided Design (ICCAD). pp. 552-559.

64. Brumley, D. and Boneh, D. Remote timing attacks are practical. *Computer Networks.* 2005, Vol. 48, 5, pp. 701-716.

65. Bar-El, Hagai. *Introduction to Side Channel Attacks: White Paper.* s.l. : Discretix Technologies Ltd.

66. Molloy, Derek. Writing a Linux Kernel Module, Part 2: A Character Device. *Linux Kernel Programming.* [Online] April 18, 2015. http://derekmolloy.ie/writing-a-linux-kernel-module-part-2-a-character-device/.

67. *An Information Theoretic Model for Adaptive Side-Channel Attacks.* B. Kopf, D. Basin. Alexandria, VA : s.n., 2017. CCS .