12-2005

Hardware Design and Implementation of Role-Based Cryptography

Scott Edward Fields
University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a thesis written by Scott Edward Fields entitled "Hardware Design and Implementation of Role-Based Cryptography." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Engineering.

Don Bouldin, Major Professor

We have read this thesis and recommend its acceptance:

Gregory Peterson, Itamar Elhanany

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

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

To the Graduate Council:

I am submitting herewith a thesis written by Scott Edward Fields entitled “Hardware Design and Implementation of Role-Based Cryptography.” I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Engineering.

Don Bouldin
Major Professor

We have read this thesis
and recommend its acceptance:

Gregory Peterson

Itamar Elhanany

Accepted for the Council:

Anne Mayhew
Vice Chancellor and
Dean of Graduate Studies

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

HARDWARE DESIGN AND IMPLEMENTATION OF ROLE-BASED CRYPTOGRAPHY

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

Scott Edward Fields
December 2005

Acknowledgments

I would like to thank all those who have helped me achieve my Master of Science degree in Computer Engineering. First, I would like to thank Dr. Don Bouldin for introducing me to FPGA design and for his continual guidance, insight, and support. Second, I would like to thank Ersin Domangue at InfoAssure, Inc. for his explanation of a number of cryptographic concepts. I would also like to thank Adam Miller and Shawn Carrithers for their co-authorship of several cryptographic modules. Finally, I would like to thank Dr. Greg Peterson and Dr. Itamar Elhanany for their many suggestions and for serving on my committee.

To my family and friends, thank you for your personal encouragement and support. Without you, this work would not have been possible.

This work was partially supported by the Office of Naval Research grant number N00014-04-1-0562 via the National Center for Advanced Secure Systems Research.

Abstract

Traditional public key cryptographic methods provide access control to sensitive data by allowing the message sender to grant a single recipient permission to read the encrypted message. The Need2Know® system (N2K) improves upon these methods by providing role-based access control. N2K defines data access permissions similar to those of a multi-user file system, but N2K strictly enforces access through cryptographic standards. Since custom hardware can efficiently implement many cryptographic algorithms and can provide additional security, N2K stands to benefit greatly from a hardware implementation. To this end, the main N2K algorithm, the Key Protection Module (KPM), is being specified in VHDL. The design is being built and tested incrementally: this first phase implements the core control logic of the KPM without integrating its cryptographic sub-modules. Both RTL simulation and formal verification are used to test the design. This is the first N2K implementation in hardware, and it promises to provide an accelerated and secured alternative to the software-based system. A hardware implementation is a necessary step toward highly secure and flexible deployments of the N2K system.

1 Introduction

This thesis addresses the hardware implementation of a new cryptographic system. The system itself aims to provide new flexibility in secure communication over insecure networks, and the implementation provides the means of successfully realizing the system under practical constraints. This implementation is the first attempt to render the core system algorithm in hardware.

This introduction examines the need for the new algorithm and discusses the practical considerations involved in implementation. Chapter 2 presents a study of relevant previous work. Chapter 3 gives a general overview of the algorithm and the design process. Chapter 4 presents the design implementation, and chapter 5 presents a discussion of the results. Finally, chapter 6 summarizes, gives conclusions, and presents possibilities for future work.

1.1 The Growing Need for Computer Security

With the increased reliance of governments, businesses, and individuals on information technology, the need for computer security is continually increasing. Modern computing practices allow data to exist in complex multi-user environments, to be transmitted via insecure public networks, and to be sent across organizational boundaries. As technology enables these new workflows, it also creates a need for new means to control sensitive information (Stallings, 2003).

1.1.1 Example 1: Military Communications

One area in need of new security measures is military communications. To pull an example from present-day politics, the U.S. is currently operating in Iraq, and its chief ally is Great Britain. A number of other countries, including Japan, have also contributed to the effort. If the U.S. decides that it is going to launch an offensive in one particular area, it needs to make sure that the allies and peace-keepers are aware of the attack, but it also needs to keep this information from the insurgents. Additionally, it might want to send in-depth operational details to the allies that should not be divulged to the peace-

keepers, or it might want to selectively deliver information only to particular allies. The current methods require that individual, direct, secure communication links be established with the involved parties. Maintaining so many secure links is a complicated and expensive operation, and a better solution should be possible.

1.1.2 Example 2: Corporate Communications

Another area that can make use of new security measures is corporate communications. Like military communications, corporate communications can involve multiple communicating entities who need access to different levels of information. For instance, the vice president of the company might want to distribute the specifications for a new product. To his manager, he might want to distribute financial forecasts along with functional specifications and packaging details. The manager might then pass along the functional specifications to his workers and outsource the packaging to a second company. Certainly, the second company should not see the functional specifications or the financial forecasts. While no harm is done if the employees see the packaging details, that is probably useless information for them. Traditionally a number of documents would be created by the vice-president, and he would have to deliver the documents to the manager with explicit instructions as to their security levels. But, this imposes the task of managing the documents on the manager, and there should be an easier, automatic way to keep track of which parties need which information.

1.1.3 Example 3: Multi-user PC File System

A third area that stands to benefit from new security measures is the multi-user PC. If the PC is running any modern multi-user operating system, then individual users can easily set permission levels to restrict access to their files. The problem with this is that those permissions are, in general, enforced by the operating system which assigns them. With relative ease, anyone with access to the physical machine can bypass the operating system and read the data stored therein. The operating system can also be bypassed by an application that takes advantage of known system weaknesses. Several available software packages make use of encryption to protect the stored data, but in

general they remove the fine-grained control that the operating system gives and limit access to a single user with the correct password. More subtle levels of access should be available with the same level of security, but there are few products that currently fill this niche.

1.2 Need2Know System

The Need2Know System was developed to address the deficiencies of other security systems and to meet the growing information security needs of complex organizations (InfoAssure N2K, 2004). Need2Know specifies an algorithm for data processing that includes the following key features:

- Centralized Role-based Access Control,
- Information Centric Security, and
- Cryptographic Enforcement.

1.2.1 Centralized Role-based Access Control

Centralized role-based access control describes the management and permissions model used by Need2Know. In this model, the top-level unit is a domain. The domain can be logically divided into a set of roles based on the flow of information within the domain, and members belong to one or more domains. A management officer oversees the domain, and adding a member means issuing credentials to the member to bind him to his associated roles. With the credentials he gains access to pertinent information, and the management officer can revoke or update the credentials in accordance with changing roles. Since management is centralized, control is maintained over the domain membership, and the system is scalable to very large organizations.

When Need2Know information is created or modified, its author can select a number of credentials to limit the readership. Selection can be as fine or as broad as desired, allowing the author to target any number of recipients, from a single individual up to the entire system. Valid permission levels are write-only and read/write, and furthermore they can be time-based, limiting the readership to members who have been with the system for some set amount of time.

1.2.2 Information Centric Security

The mantra behind Need2Know is that since *information* needs to be protected, security practices should be implemented on the *information* itself. This is known as Information Centric Security, and it improves upon previous practices that relied on physical security of networks and computer systems to protect data (InfoAssure About, 2005). Physical security still plays an important role in a total security scheme, but tying information security to the data objects themselves liberates the communication and computer systems from strict requirements. Using Information Centric Security practices, public networks and public information processing systems can often be used in the place of expensive and unruly dedicated access resources. Since Need2Know is infrastructure independent, it can ideally allow secure access to information and services nearly anytime and anywhere (InfoAssure Products, 2005).

1.2.3 Cryptographic Enforcement

Central to implementing Information Centric Security are the concepts of cryptography, which allow data to be mathematically obscured. By making use of cryptographic algorithms, data can be protected independent of the system in which it resides. Through the process of *encryption*, the original *plaintext* can be altered to produce coded *ciphertext*. With the correct digital credentials, it is possible to *decrypt* the ciphertext and obtain the original plaintext, but without them decryption can be computationally infeasible (Stallings, 2003).

With cryptographic enforcement, encrypted data can securely cross organizational boundaries, insecure channels, and various processing environments before being received and decrypted by its intended recipient(s). In contrast, a system that relies on a secure environment to enforce its access only provides information security within that environment. Once the data is passed outside of the trusted realm, for instance by being moved to a floppy disk or by being sent on the public internet, the plaintext can be viewed by any interested party.

The Need2Know system can only be as secure as its underlying cryptography, so choosing good algorithms is a key concern. In the security community, cryptographic

algorithms are heavily scrutinized to determine their level of security. In the U.S., a number of organizations publish cryptographic standards that have been approved for widespread public (government) and private (corporate) use. The main standards bodies are the National Institute of Standards (NIST) and American National Standards Institute (ANSI), and the Need2Know system makes use of their published standards wherever they are applicable.

1.2.4 Enabling New Capabilities

With the aforementioned features, the Need2Know system gives new information security capabilities to its users, enabling more-flexible access to shared secure information. With regards to the military communications example, wartime updates of U.S. troop movements could be sent over public wireless or internet channels, even being broadcast at a known frequency and time. Despite the fact that the transmission could be intercepted by any party, only those nations with the correct permission levels would be able to access the privileged information. For the corporate communications example, one set of documents could be distributed with each document being protected against unauthorized use. Then, even if the outsourced company or the engineers received the marketing data, they would not be able to read it. Finally, with regards to the multi-user PC example, files would be assigned permissions based on Need2Know rather than the operating system. The same fine-grain access permissions would be possible, but in cases when the operating system was replaced, bypassed, or forgotten, the files would still maintain their intended access controls.

1.3 Implementation Considerations

In Need2Know, the main algorithm resides in the Key Protection Module (KPM). The KPM makes use of the various cryptographic standards in order to encrypt and decrypt Need2Know data. Therefore, each Need2Know member needs an implementation of the KPM in order to interact with the system. Need2Know, by design, has great potential to allow flexible, secure access to information. However, to realize this potential, the implementation must meet several requirements:

- low time-to-market,
- high-throughput,
- low power,
- low cost,
- feature flexibility, and
- implementation security.

In making implementation choices it is necessary to evaluate the tradeoffs among these metrics.

1.3.1 Low Time-to-Market

Design-time is critical because Need2Know aims to meet a current need, not a need that is only projected to exist in the future. With a fast turn-around, development costs for the implementation are kept down. The sooner Need2Know is ready to be deployed, the sooner it can benefit its users and the sooner its developers can earn a return on their investment.

1.3.2 High-Throughput

High-throughput refers to how quickly the implementation can encrypt and decrypt information. The two contributing factors to throughput will be the number of recipient roles and the size of the data to be encrypted. For every additional recipient role, an extra set of cryptographic operations will need to be performed, and increasing data size will necessarily extend the length of the encryption/decryption operation. Regardless of the source of the latency, a processing time that is too great negates the usefulness of the device. Often, the need to share information is time-critical, many users need to be targeted, and large amounts of data need to be transferred.

1.3.3 Low Power

Low power becomes of interest when taking into account mobile, battery-life constrained devices such as radios or laptop computers. For these devices, a power-intensive implementation can limit or even remove usefulness of the implementation.

Additionally, power consumption generates heat, and the need for active cooling can radically affect a device's design and viability (Adams, 2002). Certainly, if the implementation has the power and cooling requirements of a modern desktop CPU, then the mobility and installation requirements are limited.

1.3.4 Low Cost

A low product cost is a concern of any commercial device, and the KPM is no exception. With a high cost, maximum performance can be realized, but the product might be unaffordable. Conversely, an ultra low cost bolsters widespread distribution, but low performance might make the product undesirable. Upfront non-recurring engineering costs (NREs) can be an important factor along with per-unit costs, and the relative tradeoffs need to be evaluated in the light of the business plan (Adams, 2002).

1.3.5 Feature Flexibility

Feature flexibility is an important consideration given that during the design cycle or after deployment the changes might be required in the implementation. Since the Need2Know specification is relatively new, it is reasonably likely that changes or clarifications in the specifications will be issued. Also, the core cryptographic algorithms are continually being analyzed and refined, so it is important to be able to support new developments to maintain the security of the implementation.

1.3.6 Implementation Security

Even though the Need2Know algorithm uses cryptographic standards to provide strong security during data transmission, data and secret information can exist in relatively unprotected forms within the implementation. If a potential attacker believes that the implementation is a weak point in the security scheme, he will focus his efforts on attacking it. While it is virtually impossible to guarantee security of a device, a number of implementation decisions can improve the device's physical security.

1.4 Technology Tradeoffs

Three technologies were considered for this implementation: software running on a general purpose CPU, Field Programmable Gate Arrays (FPGAs), and Application Specific Integrated Circuits (ASICs). With any engineering design, it is difficult to find a single technology that optimizes every constraint. The relative tradeoffs are evaluated against each other in the following subsections.

1.4.1 Low Time-to-Market

Software provides the fastest migration from conceptualization to implementation. Software easily leverages existing libraries of functionality and can be quickly debugged and deployed on off-the-shelf PC hardware. FPGA and ASIC designs can also make use of existing designs, but integrating legacy functionality generally requires more extensive testing and debugging. Also, since FPGA and ASIC developments take place at a lower level of abstraction than software development, development and debug cycles are longer. ASIC prototyping further entails a multi-week to multi-month wait while the device is fabricated, whereas FPGAs designs offer rapid prototyping.

1.4.2 High-Throughput

While either large number of recipient roles or large message size can dominate processing time, both of the involved operations make heavy use of cryptography. Software is relatively inefficient when dealing with cryptographic algorithms, but those algorithms can be very efficiently implemented in hardware. Typically, FPGAs and ASICS can offer an order of magnitude increase in performance compared to general purpose processors. For a given design, ASICs are generally able to offer higher clock speeds than FPGAs, so their scalability will also be noticeably higher.

1.4.3 Low Power

Modern general purpose processors needed to run software employ high clock frequencies in the gigahertz (10^9) range, whereas FPGAs and ASICs use lower clock

frequencies of tens or hundreds of megahertz (10^6). Hardware power dissipation scales linearly with clock frequency, and ASIC implementations fully realize this potential power savings: they can be expected to consume $1/10^{th}$ to $1/100^{th}$ the power of their general purpose counterparts. FPGAs, on the other hand, are unable to realize low power implementations due to higher transistor counts and higher internal losses. While their power usage is much less than that of a mainstream desktop CPU, it is often still too high for power-conscious mobile applications.

1.4.4 Low Cost

While high-end general purpose processors are extremely expensive, their embedded counterparts are priced much more reasonably. As these processors are off-the-shelf components, they have no additional custom mask costs. FPGAs offer the same performance as a much more expensive general purpose CPU at a lower per-unit cost while still remaining free of extra costs. ASICs, while offering the best per-unit cost of the three, have associated mask costs that can soar into the millions-of-dollars range. ASICs thus offer the lowest total cost only when amortizing mask costs over high production volumes.

1.4.5 Feature Flexibility

Software offers the ultimate in feature flexibility: most feature updates can be instituted with an update to the code. Thus, changes can be made throughout the development cycle and after deployment. In a similar manner, FPGA designs can be updated by altering the configuration memory, though it is possible that the new features cannot be accommodated by the available logic resources. In contrast to software and FPGA implementations, ASICs provide absolutely no feature flexibility, even during the design debug phase.

1.4.6 Implementation Security

From a security standpoint, a software implementation is the least secure. In an OS environment, there are a number of opportunities for rogue programs and users to snoop on activity of other programs and data. Even in a limited software environment,

the system is more open to attack through altered programs or through analysis of the well-understood hardware. The dedicated hardware of FPGAs and ASICs provides an extra layer of security since access to the computing resources can be more closely controlled. FPGAs, allowing reconfiguration, have a potential weakness that ASICs lack; the possibility exists that they can be surreptitiously reconfigured to open vulnerabilities in the system, though admittedly this is a much more involved procedure than an attack on a software implementation.

1.5 FPGA Implementation

After weighing the tradeoffs, an FPGA implementation was chosen. The implementation needs to provide a computational base for high mobility, high performance electronic devices for use with the Need2Know system. Hardware is necessary to realize the goal of providing secure access to information anywhere, anytime, and the FPGA technology provides a balance of high performance and flexibility necessary for prototyping. Choosing an FPGA implementation also gives a clear migration path to an ASIC implementation, should the potential benefits of that technology be desired in the future.

This document describes in full the FPGA implementation of the Need2Know KPM. Organization is as follows: Chapter 2 explores the previous work on cryptographic coprocessors. Chapter 3 gives in-depth coverage of the KPM algorithm and FPGA design steps. Chapter 4 describes the logic design of the implementation. Chapter 5 discusses the implementation results. Chapter 6 concludes this work and outlines possible directions for future work.

2 Background Study

The need for cryptographic coprocessors is widely addressed in literature, both by academic researchers and by commercial designers. As per its name, a coprocessor works in conjunction with some other microprocessor or microcontroller to produce a desired result. Thus, the major design consideration for a coprocessor is how much of the processing to offload from the host processor. In making this choice, there is a tradeoff between flexibility and performance. The explored design strategies for cryptographic coprocessors are, in decreasing order of flexibility:

- Programmable coprocessors,
- Reconfigurable coprocessors,
- loosely-integrated coprocessors, and
- tightly-integrated coprocessors.

2.1 Programmable Cryptographic Coprocessors

Programmable cryptographic coprocessors offer the highest level of flexibility. Like any programmable processor, they implement some basic set of operations and provide these as sequentially-linked building blocks for complex algorithms. A coprocessor's design differs from that of a general purpose processor because it is assumed that only certain types of algorithms will be run. This being the case, the instruction set architecture (ISA), data path logic, and control logic can be optimized for a limited selection of workloads. To differentiate specialized programmable processors from general purpose processors, they are sometimes called domain-specific processors (Hodjat and Verbauwhede, 2004).

2.1.1 Generalized Functional Units

One technique used is that of generalized functional units. In this case, the processor's functional units do not single out any particular algorithm. Rather, they perform operations applicable to a class of algorithms. A selection of algorithms, e.g. a number of symmetric encryption algorithms, is first analyzed to determine the most

common operations. These operations are then given the highest value through the design, but the functional units must still maintain enough functionality to be able to compute general purpose results, consuming some processing overhead. Obviously, such a processor will perform much better on those algorithms that were considered by its designers, so it is important to note what considerations went into the design.

CryptoManiac is an architecture that makes use of generalized functional units (Wu, Weaver, and Austin, 2001). The design is a 4-wide 32-bit Very Long Instruction Word (VLIW) architecture with a 4-stage pipeline, as shown in Figure 1. Each word contains up to four independent instructions that can be computed in parallel. There is no cache, and a simple branch target buffer (BTB) is used in the branch predictor.

The functional units are optimized for symmetric key encryption algorithms. In particular, the following algorithms were analyzed during the design stages: Blowfish, 3DES, IDEA, Mars, RC4, RC6, Rijndael, and Twofish. The resulting analysis showed commonalities among the algorithms, and the design was optimized to reflect these. As shown in Figure 2, the units make use of a multiplier, a substitution box (SBOX), an adder, a rotator, and two logical units.

The CryptoManiac processor supports a number of concurrent encryption sessions with separate keys and data. Each VLIW processor is called a processing element, and a number of these processing units are used within CryptoManiac. Input requests are loaded into a shared input queue and processed by a scheduler. The scheduler directs each instruction to a processing unit, which loads its session info from a key store. Finally, results are deposited in a shared output queue.

Another coprocessor based on generalized functional units is Cryptonite (Butchy, 2002). The stated mindset of the processor's designer was to "not be a collection of specialized hardware. Instead, it should be based on ideally primitive and reusable hardware functions." Just as with the CryptoManiac, a number of symmetric encryption algorithms were analyzed (DES, AES, SHA, MD5, IDEA, and RC6) and the common operations were determined.

The Cryptonite architecture, while being VLIW, differs significantly from that of CryptoManiac. Cryptonite uses a 3-stage pipeline with fetch, decode, and ex/writeback.

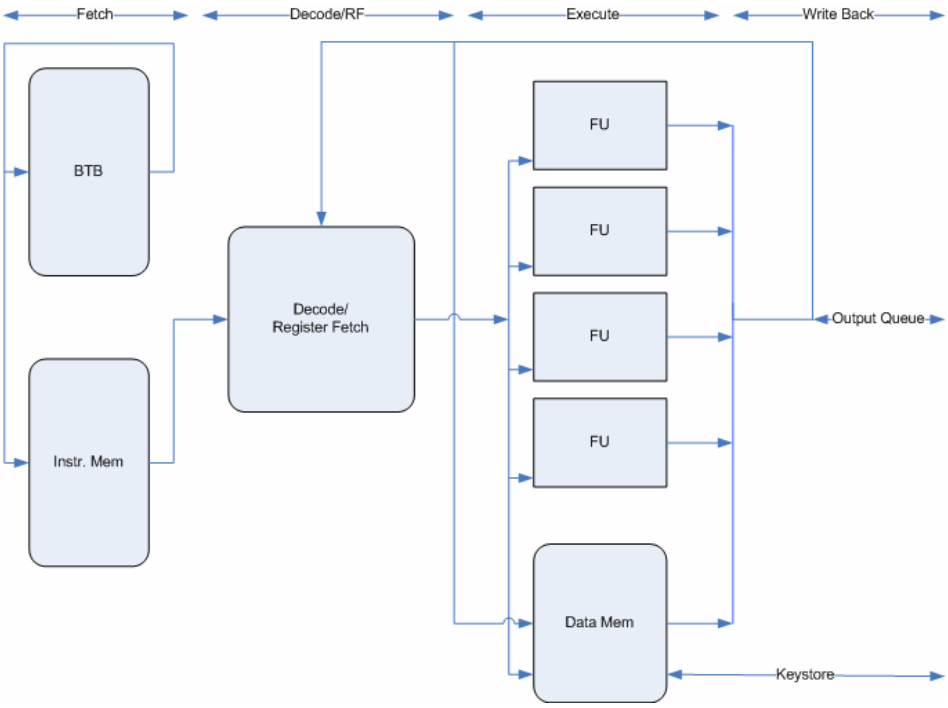

Figure 1: The CrypoManiac processing architecture uses generalized functional units.

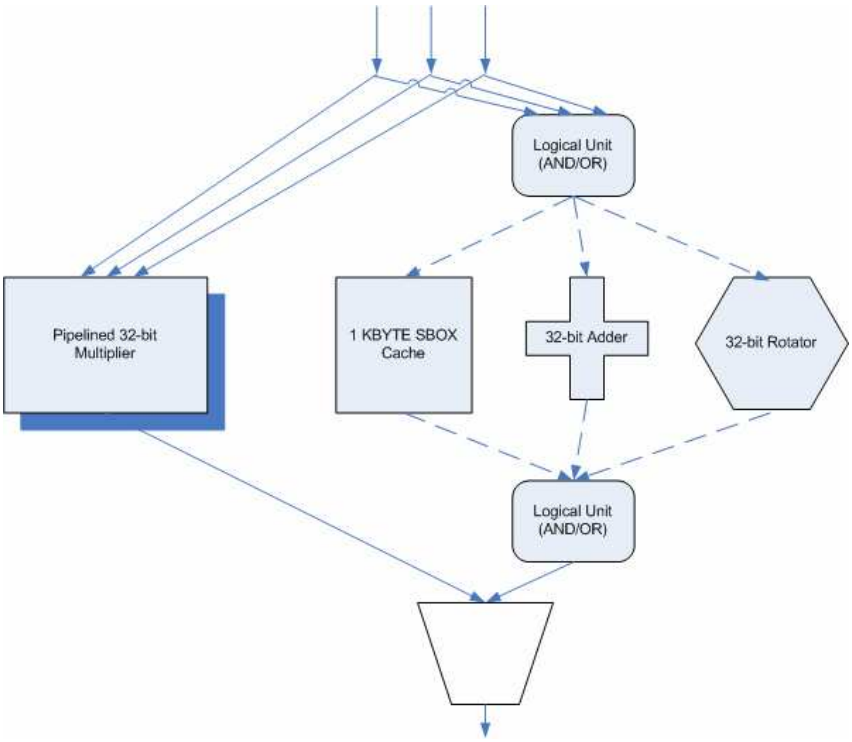

Figure 2: The CryptoManiac functional unit is optimized for cryptographic applications.

It makes a strict distinction between data path and control, maintaining the SBOX data outside the functional units, unlike the CryptoManiac design. Also, while the Cryptonite architecture supports two execution paths, it makes use of two separate memories to feed the two separate ALUs. The design uses 64-bit registers, and the execution cycle is strictly single-cycle, unlike CryptoManiac's pipelined multipliers.

The generalized approach has also been applied to asymmetric algorithms. Researchers at Sun have created a VLIW public key coprocessor for RSA and ECC (Eberle, et al., 2004). This processor makes use of optimized 64-bit multipliers to accelerate the multiple precision operations common in asymmetric cryptography. Also of note, each instruction word is split into two regions – one for a control operation and its operands and one for a data operation and operands.

2.1.2 Integrated Specialized Functional Units

Another approach to accelerating cryptographic operations is to start with a general purpose processor and add custom instructions to the ISA (Ravi, et al., 2002). This is accomplished by integrating specialized functional units into the existing pipeline. When custom instructions travel down the processor pipeline, the specialized functional units take the place of the general purpose functional unit to execute them. Since the fetch, decode, and pipeline logic is shared between the general purpose and specialized functional units, this scheme efficiently reuses hardware resources. An architecture with custom DSP, security, and networking functional units is shown in Figure 3.

The design was based on the Tensilica Xtensa T1040, a configurable 32-bit RISC processor. First, the authors built a set of software libraries to implement the symmetric algorithms DES, 3DES, and AES as well as the asymmetric algorithms RSA and El-Gamal. Care was given to specify the libraries in a hierarchical fashion with high-level algorithms composed of low-level functions. Then, by profiling the libraries, it was possible to determine which functions would benefit most from hardware acceleration. The hardware design space was iteratively explored as follows: the functions of interest were hand-optimized as hardware implementations, the processor was profiled, and

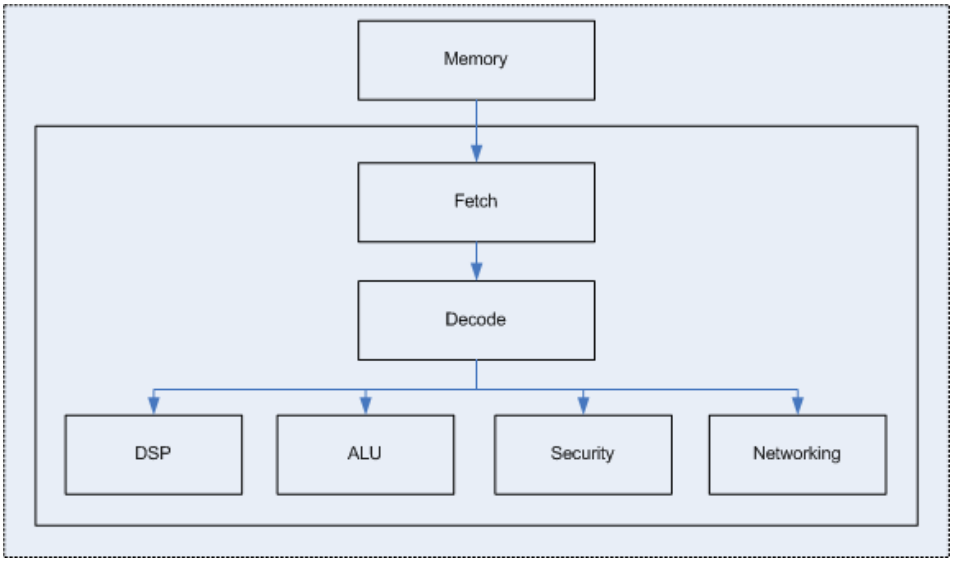

Figure 3: Various specialized functional units can be integrated into the processing.

overall area and delay were inspected. These steps were repeated, resulting in a final design that met the performance goals.

2.1.3 External Specialized Functional Units

For coarse-grained hardware acceleration, it is necessary to move the specialized functional units outside the main CPU's pipeline. This scheme uses loosely-coupled independent coprocessors that connect to the main CPU over a dedicated interface. Since the coprocessors are outside the main pipeline, they can be larger and can perform more complex operations. The downside is that this organization incurs some overhead for the interface and data buffering. A high-level view of this type is shown in Figure 4, in which a main CPU has been linked to DSP, security, and networking coprocessors.

A programmable AES coprocessor was designed with this organization in mind, as shown in Figure 5 (Hodjat and Verbauwhede, 2004). Unlike the previously discussed programmable architectures, this one makes use of coarse instructions that are specific to a single algorithm. Instructions from the CPU are received over a memory mapped interface, giving the CPU direct access to the coprocessor's two 8-bit instruction and configuration registers and two 32-bit input and output registers. Two categories of

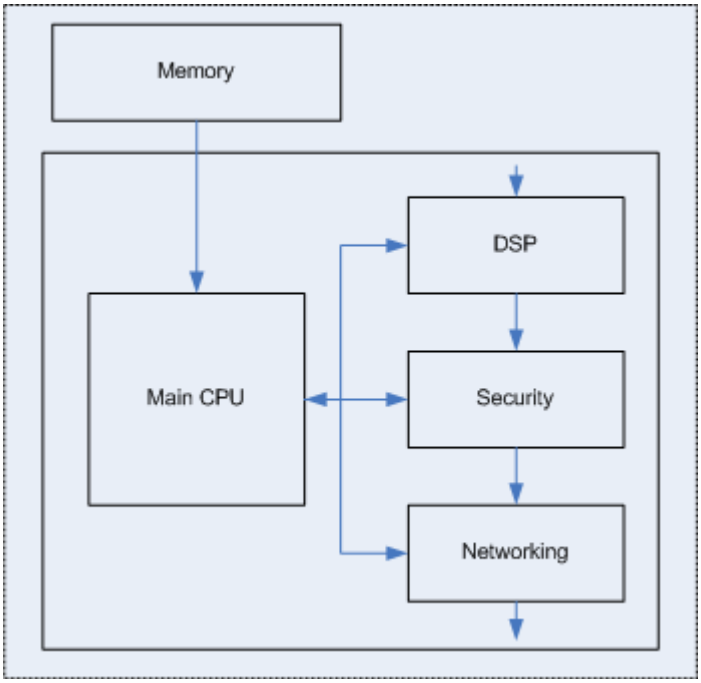

Figure 4: Specialized functional units are attached external to the main CPU.

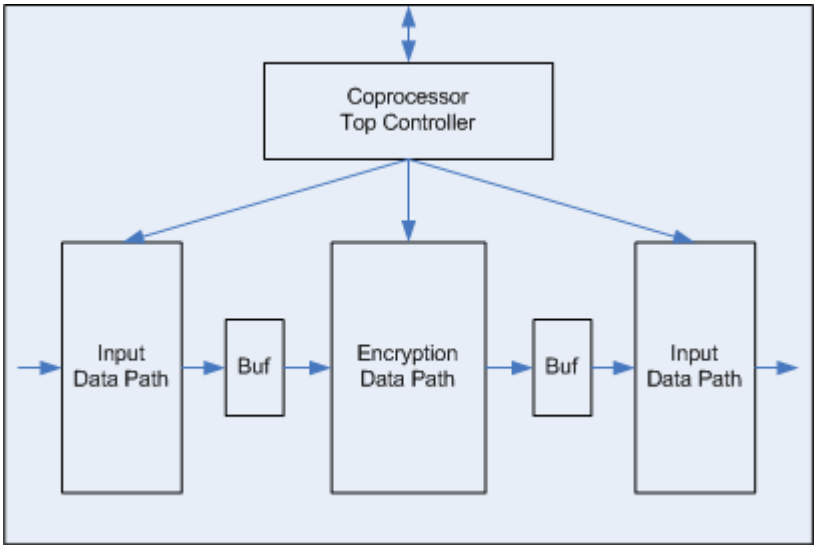

Figure 5: An external specialized functional unit contains its own input, output, and control logic.

instructions are used: single and continuous. While single operations are standard, continuous instructions facilitate operations on streams of data.

The coprocessor internals are modular, comprised of an input module, output module, encryption module, and top controller. Each module incorporates a state machine, and the machines are linked in a hierarchical fashion. The design uses a three-stage block pipeline, reading one block of data into the input stage, processing one block in the encryption stage, and outputting one block via the output stage. To match the pipeline cycle count to the slowest stage, each stage of the pipeline takes 11 clock cycles to complete.

2.2 Reconfigurable Cryptographic Coprocessors

In addition to flexibility through programmability, a hardware coprocessor can achieve flexibility through reconfiguration. In general, these solutions maintain a store of encryption algorithms and can dynamically load any of the supported algorithms into the device in order to complete a request. The controlling CPU is responsible for selecting the desired algorithm.

Reconfiguration offers a couple of distinct advantages over a fixed design. First, a number of modern security standards, such as the Secure Socket Layer (SSL) and Internet Protocol Secure (IPSec), define frameworks in which to use various symmetric or asymmetric algorithms. Using reconfiguration, the required algorithm can be dynamically paged into the coprocessor, and then the controlling CPU can tie various algorithms together into some high-level standard. This results in an area savings since the peak number of gates required is the number of gates required by the single largest algorithm. Second, since the configuration memory can be reprogrammed, the hardware can be field-upgradeable. This is a useful feature when algorithms need to be updated or replaced to handle changing standards or to combat cryptographic attacks.

Since reconfigurable coprocessors generally support only one encryption algorithm at a time, they cannot take advantage of hardware's potential for parallelism. Also, since they make use of FPGAs rather than fixed gates, they generally have higher power consumption, lower throughput, and higher per-unit cost.

2.2.1 Whole-FPGA Reconfiguration

One approach to reconfiguration is to reconfigure the entire FPGA when a new encryption algorithm is required. One such architecture, an Algorithm-Agile Cryptographic Coprocessor, is shown in Figure 6 (Paar, Chetwynd, Connor, Deng, Marchant, 1999). In order to interact with the host PC, a number of external blocks are used, including a system controller and an algorithm library processor. The algorithm library processor is responsible for paging entire FPGA bit-streams into the FPGA.

This architecture focused on implementing the symmetric encryption AES candidates. While the input, output, and control blocks are the same for any bit-stream, there is a custom algorithm core. The architecture makes use of a standard interface to each of the custom cryptographic cores. The number of supported algorithms is limited only by the algorithm library, and this architecture's library has a seven-algorithm capacity. A similar reconfigurable architecture with minor updates was explored in (Mingyu, Jinahua, Guangwei, 2003).

Another architecture, the Adaptive Cryptographic Engine (ACE) shown in Figure 7, aggressively addresses the storage limitations of the algorithm library (Dandalis and Prasanna, 2000). The configuration bit-streams are organized in memory as parameterized configuration skeletons and various parameter sets. The skeletons and parameters are combined at run time to generate the bit-stream for device configuration. Furthermore, a variant of LZ compression suitable for hardware is used to realize compression ratios of 65%-95%.

2.2.2 Partial Reconfiguration

The CryptoBooster architecture shown in Figure 8 makes use of dynamic partial reconfiguration, reconfiguring only a portion of the configurable gates when a new algorithm is required (Mosanya, 1999). This has two potential advantages over whole-FPGA reconfiguration: algorithm-switching can be faster and configuration storage requirements are lower.

Because part of the processor fabric can be replaced on demand, the architecture must be highly modular. This modularity is enforced through the use of a standard point-

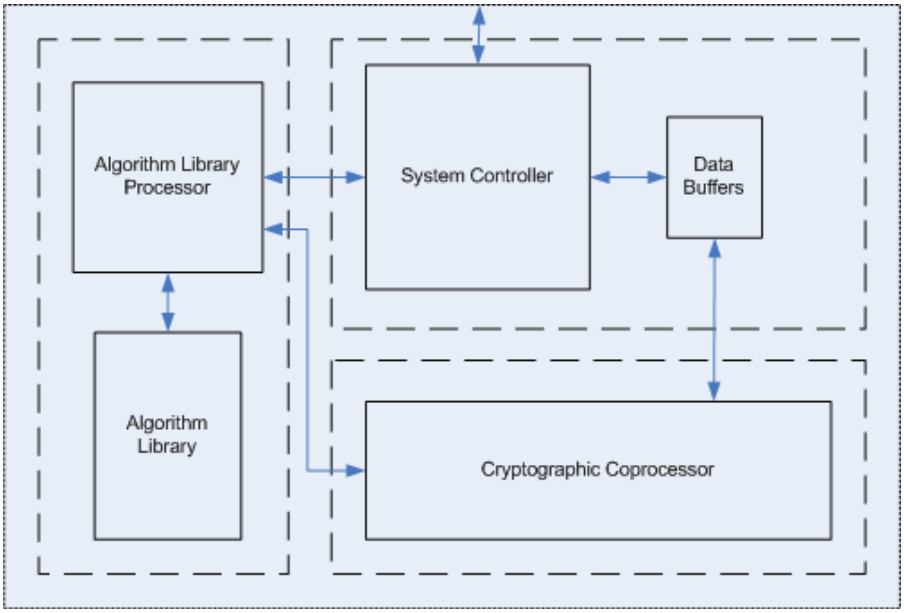

Figure 6: A processor, controller, and coprocessor comprise the algorithm-agile cryptographic coprocessor.

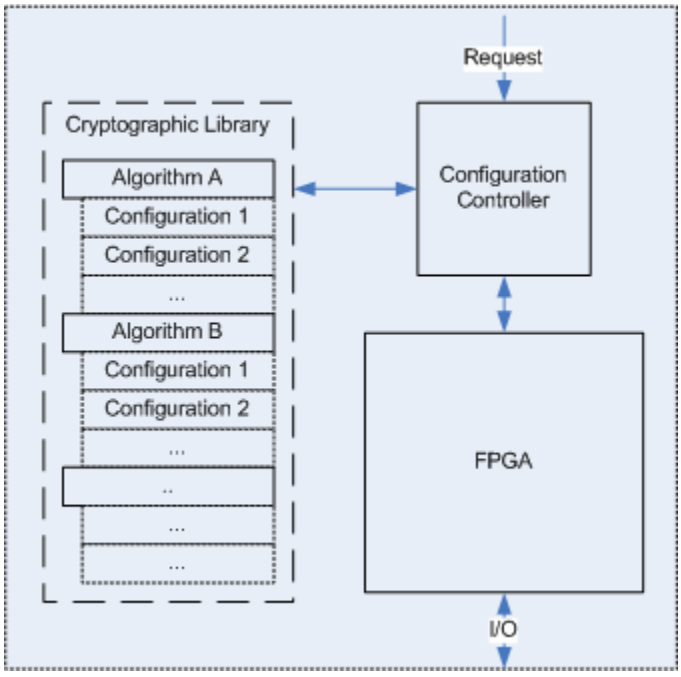

Figure 7: The adaptive cryptographic engine uses dynamic bit-stream synthesis.

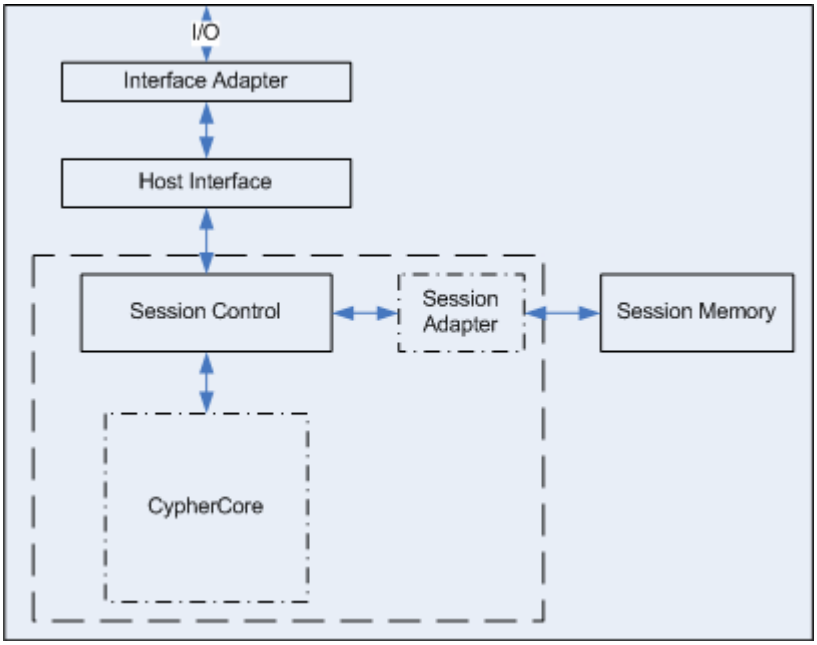

Figure 8: CryptoBooster's architecture allows dynamic reconfiguration of the encryption algorithm.

to-point interface between the encryption algorithm module, or CypherCore, and the session control module. The standard communication signals support a query/response system, allowing the "intelligent" algorithm module to reports its capabilities to the controller. Queries and responses are packets of control or data information.

The architecture also supports multiple encryption sessions, requiring a session adapter module and access to session memory. The session adapter is specific to the encryption algorithm, so it, too, is dynamically reconfigured when a new encryption algorithm is loaded. A query/response point-to-point link also exists between the session control module and the session adapter.

2.3 Loosely-Integrated Cryptographic Coprocessor Systems

Fixed cryptographic coprocessors can be divided based on the level of integration of their algorithms. In loosely-integrated systems, a number of algorithms are supported, but the host CPU is responsible for tying their results together for use in a high-level protocol. Various approaches have made use of FPGA or fixed-gate technologies. Both

provide higher performance compared to the reconfigurable cryptographic coprocessors since they remove the latency of reconfiguration. However, this benefit comes at the expense of increased area needed to support multiple algorithms.

2.3.1 Partial Protocol Support

Modern communications protocols, such as IPSec and TLS, define support for numerous cryptographic algorithms, but some coprocessor architectures offer only partial protocol support. This approach lessens the complexity of the design and reduces chip area, but it is most useful if only a part of the protocol will be needed or if only the most common case needs to be sped up. With the reduced area requirements, these designs can be implemented in an FPGA.

An architecture that supports AES and SHA/HMAC for the IPSec protocol is shown in Figure 9 (McLoone and MCanny, 2002). The two cores can operate in parallel, providing an increase in the encryption performance. As shown in the figure, the architecture contains a minimum of control, and the host CPU is responsible for

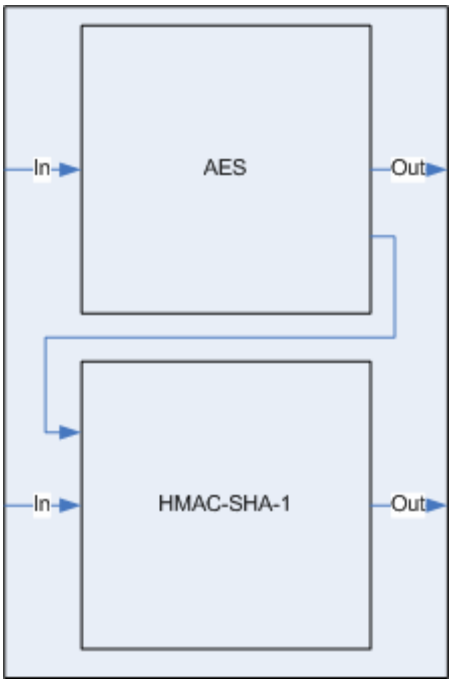

Figure 9: This loosely-integrated architecture offers AES (Rijndael) and HMAC-SHA-1 functionality with a minimum of control logic.

providing all the setup and control signals and incorporating the results into some high-level protocol. This design was tested in a Xilinx Virtex XCV1000E FPGA.

Another architecture that has partial protocol support is outlined in Crowe, Daily, Kerins, and Marnane, 2004. This architecture supports AES, SHA-512, and RSA DSA, and it is designed to accelerate IPSec and TLS communications. Unlike the previous architecture, this design offers public key algorithm acceleration along with symmetric key operations. This design, as with the previous, allows the AES encryption block to run in parallel with the other blocks.

In this architecture, the designers attempted to optimize whole-chip performance rather than the performance of a single algorithm. One technique they used was to insert a FIFO buffer between the shared memory interface and the algorithm blocks (both on the input and output sides). This allows the symmetric encryption and the SHA/DSA operations to run in parallel from the same input source. The architecture also makes use of multiple clock domains, allowing the RAM and logic to run at different rates. This design was implemented and tested on a Xilinx Virtex XCV2000E FPGA.

2.3.2 Full Protocol Support

Loosely-integrated architectures with full protocol support implement all of the functions needed by the high-level protocol. In essence, they connect a number of algorithm-specific cores on a system bus and provide the various core functions to the host CPU. These architectures have high area requirements to support the various algorithms, and the bus is a necessity to handle the added complexity of so many cores. In the interest of increasing throughput, they might incorporate supporting algorithms such as random number generators, and in the interest of increasing security, they might implement attack resistance measures.

The Discretix CryptoCell, shown in Figure 10, provides full protocol support (Discretix, 2005). A number of algorithm-specific cores are implemented on a bus. The symmetric key operations of AES, DES, and 3DES are supported, the hash operations of SHA1, MD5, and HMAC are supported, and the public key operations of RSA, DSA, ECC, and DH are supported. In order to implement the IPSec protocol, or any other

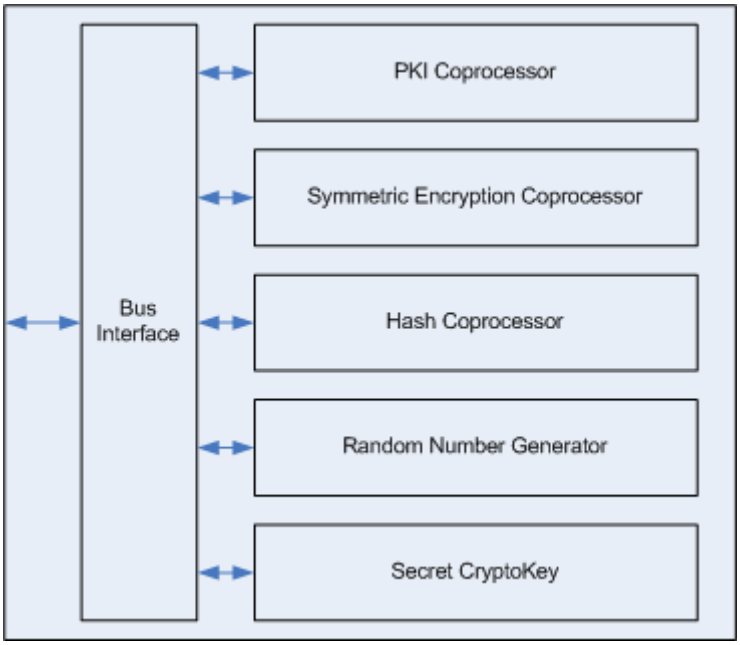

Figure 10: The Discretix CryptoCell is a loosely-integrated coprocessor with full protocol support.

high-level protocol, the host CPU ties together the results of the various cores. Additionally, a random number generator is included, as are unspecified attack resistance components.

IBM offers a similar full protocol coprocessor, the UltraCypher Cryptographic Engine (IBM, 1998). The UltraCypher provides specialized cores for DES, 3DES, MAC, 3MAC, RSA, modular exponentiation, modular arithmetic, SHA1, and random number generation. Its interface is via the ISA bus, and the host CPU has read/write access to various control, setup, and status registers. Data buffering is via FIFO buffers.

2.4 Tightly-Integrated Cryptographic Coprocessors

Tightly-integrated coprocessors offer the hardware acceleration. Instead of relying on the host-CPU to direct the links among various cryptographic operations, link some or all of these operations using internal control. The input to these processors can be entire communication packets, rather than raw data, and the processor can be responsible for parsing the header and trailer, verifying and decoding the payload, and creating the output packet. Rather than necessarily being more restrictive, these

coprocessors can offer a superset of the loosely-integrated functionality. Some or all of the packet processing functionality can be revealed to the host CPU, resulting in the same level of access to low-level cryptographic functions.

The SafeNet SafeXcel-1840 coprocessor, shown in Figure 11, uses the tightly-integrated approach (SafeNet, 2005). Its overall architecture links a public key accelerator with a number of packet processing engines via a bus. Each packet processing engine incorporates a symmetric encryption algorithm and a hashing algorithm, and along with those are input/output buffers, header and trailer parsing logic, context memory, and control logic.

The SafeNet coprocessor supports AES, DES, 3DES, and ARC4 symmetric encryption algorithms, MD5 and SHA1 hashing algorithms, and DH, RSA, and DSA public key algorithms. A random number generator is also included. For interfacing, PCI, PCI-express, or a shared memory interface are supported. Since the packet engines and public key accelerators are separate entities, they can run in parallel for a performance boost. Performance is also increased because a dedicated hardware controller chooses which engine receives which input packet, communication with the host CPU is packet-based, and burst packet transfers are possible.

Another tightly-integrated coprocessor is the Hifn HIPP 7855 (Hifn, 2005). Feature-wise, this coprocessor is similar to the SafeNet design, but the architectural details are not made available to the public.

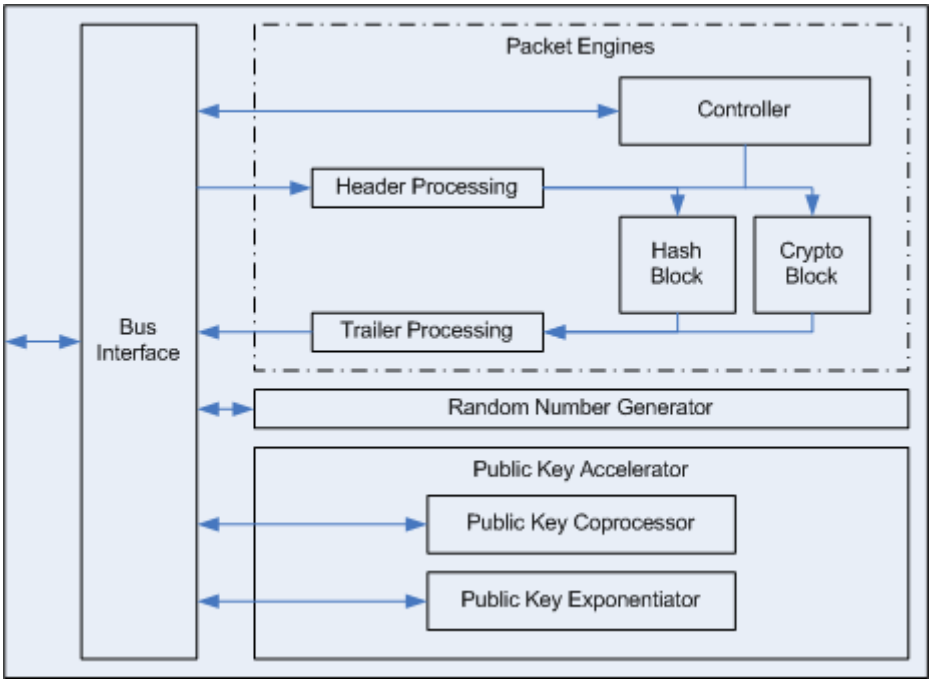

Figure 11: The SafeNet SafeXcel-1840 is a tightly-integrated coprocessor with packet processing.

3 General

In the N2K, the KPM specifies how various cryptographic standards are linked to provide cryptographic Role Based Access Control (RBAC). The following sections are organized as follows. First, the basic N2K organization is presented. Next, the various steps of the KPM, as well as their associated cryptographic standards, are elaborated in the order that they are conducted. Finally, the FPGA design flow is presented.

3.1 N2K Organization

N2K has a carefully defined centralized management scheme that enforces separation of duty. That is, a number of management roles are defined, and no one role has the power to compromise the integrity of information protected by N2K. The details of N2K administrative roles, while important, have little to do with the KPM and are not discussed here (InfoAssure N2K, 2004). Additional administrative steps, while mentioned, are not discussed in detail.

Administrators oversee cryptographic domains, many of which might be represented within a single organization. A domain defines the elliptic curve math to be used by specifying the curve, base polynomial, and base point. The NIST-approved K-571 and B-571 curves are the only approved curves, but the base polynomial and base point are generated by the administrators.

Access control in N2K is based upon possession of label key pairs, or labels. Labels are pre-computed elliptic curve Diffie-Hellman (ECDH) key pairs coupled with a globally-unique identifier, or GUID. For ECDH, the public key is a point on an elliptic curve, and the private key is a large integer. When data is encrypted, a number of labels are assigned to set the access permissions. Possession of the required public keys gives a user write access, while possession of the required private keys gives a user read/write access. Except for the responsibility of maintaining private key secrecy, the security of the system is enforced via cryptographic means.

It is the responsibility of the N2K administrators to generate and distribute the labels. Also, at any time, the administrators can update labels, enabling access to be

temporally restricted. Another duty of the administration is to select appropriate cryptographic algorithms used in the scheme. This choice is made according to the organization's needs, and for the rest of the discussion it is assumed that a subset of the available options has been selected.

3.2 KPM Encryption

The KPM encryption process is the sequence of steps necessary to create an N2K packet. A user must select the desired access permissions and create the initial content, and then these inputs are run through the KPM algorithm to produce the encrypted output. The process of data encryption is as follows:

1) The user chooses labels (named key pairs) that determine permissions of the encrypted data.
2) A valid symmetric key (*K*) and initialization vector (*IV*) are generated randomly.
3) An ephemeral private key (d_e) for each domain is generated.
4) A public key ($\boldsymbol{Q_e}$) is calculated for each ephemeral private key.
5) For each label set, a key encryption key (*KEK*) is derived.
6) For each label set, *K* is wrapped with a *KEK* to generate the wrapped key (*wK*).
7) The data (*P*) is signed with the user's private key (d_s) to generate the inner signature (*S*).
8) *P* is encrypted with *K* and *IV* to generate the encrypted data (*C*).
9) The metadata and encrypted data are packaged.
10) The package is signed with the user's private key (d_s) to generate the outer signature (*SH*). This binds the encrypted message and the metadata and finalizes the N2K packet.

A top-level view of the KPM encryption process is shown in Figure 12, and the various steps are discussed in more detail in the following sections.

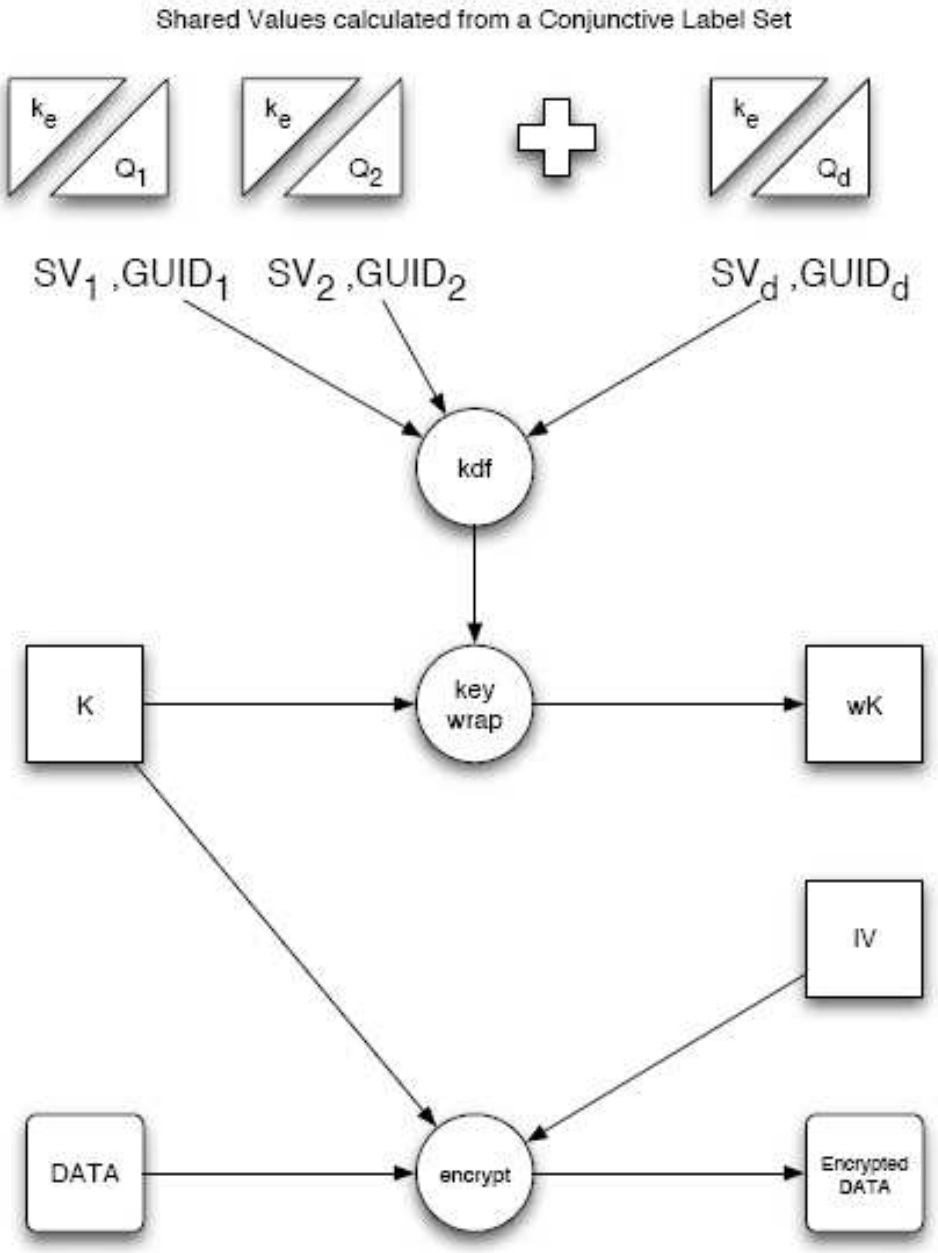

Figure 12: Label sets, random values, and data are used to generate encrypted data and header information (InfoAssure N2K, 2004).

3.3 KPM Decryption

The KPM decryption process is the reverse of the encryption process. When an authorized user receives the packet, he or she will be able decrypt the packet and regenerate the original input content. The KPM data decryption steps are as follows:

1) The encrypting user's public key ($\boldsymbol{Q_s}$) is used to verify the outer signature (*SH*).
2) A shared values (*SV*) is calculated for the label for which the recipient holds private keys.
3) From the set of *SV*, *KEK* is derived.
4) *KEK* is used to unwrap *wK* and recover *K*.
5) *K* and *IV* are used to decrypt the data *P*.
6) Using the encrypting user's public key ($\boldsymbol{Q_s}$), the inner signature (*S*) is verified against the decrypted text.

A top-level view of the KPM decryption process is shown in Figure 13, and the various steps are discussed in more detail in the following sections.

3.4 KPM Keying Material

Various keys are used throughout the N2K process. Some of this keying material is generated by administrative processes outside the scope of the KPM, such as the label keys. The KPM itself, however, is responsible for generating data encryption keys, initialization vectors, and ephemeral key pairs. Random values are needed for all of these parameters, and secure random number generation is essential for the security of the scheme. Pseudo-random number generation standards are employed, but they must be seeded with a high-quality source of entropy.

3.4.1 Random Values

A NIST-approved pseudo-random number generator from ANS X9.63, Annex A.4.1, is used for the KPM (Accredited Standards Committee ANS X9.63, 2002). This generator is seeded with a non-deterministic random number and then relies on successive applications of the cryptographic hash algorithm SHA-1 to generate its values. The pseudo-random number generation function is defined as

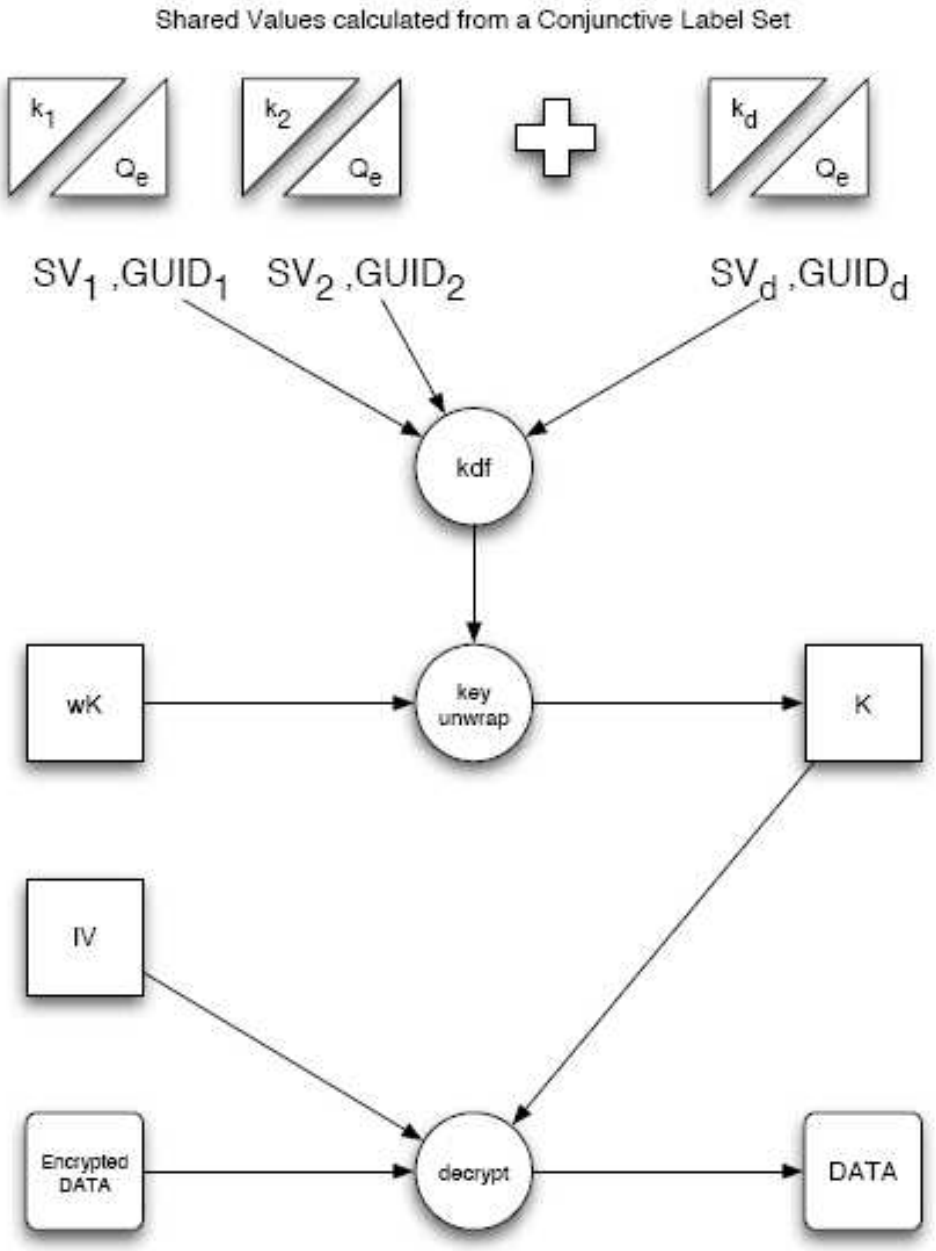

Figure 13: A label set, the header, and the encrypted data are decrypted to recover the original data (InfoAssure N2K, 2004).

$$png(l, n, x, y)$$

where the output is l random values, each less than n. In general, n is $2^b - 1$, where b is the number of bits needed for the random values. The x and y values are both b-bit real random values used for seeding the generator.

3.4.2 Ephemeral Key Pairs

During encryption, ephemeral key pairs are generated for each domain spanned by the relevant label set. The ephemeral private key, d_e, is generated with the pseudorandom number generator such that

$$d_e = png(1, r, XKEY, XSEED)$$

where r is the order of the elliptic curve and the *XKEY* and *XEED* values are generated randomly. The ephemeral public key, $\boldsymbol{Q}_e$, is then derived from the ephemeral public key via

$$\boldsymbol{Q}_e = d_e\boldsymbol{G}$$

where $\boldsymbol{G}$ is the domain base point. Note that since both $\boldsymbol{Q}_e$ and $\boldsymbol{G}$ are points, they are represented with bold type.

3.5 KPM Labels and Label Combining Logic

To assign access permissions to data, a content creator must select some number of labels. These labels can be combined using the logical "AND" and "OR" operators to narrow or broaden permissions, respectively.

3.5.1 Disjunctive and Conjunctive Label Sets

Disjunctive Normal Form (DNF) is the standard form for entering label sets at the user or application level. In this form, labels are combined with "OR" operators to form a label set, and multiple label sets can be combined with "AND" operators. An example is:

$$L_1 \text{ AND } (L_{2,1} \text{ OR } L_{2,2} \text{ OR } \ldots) \text{ AND } (L_{3,1} \text{ OR } L_{3,2} \text{ OR } \ldots) \text{ AND } \ldots$$

where a label $L_{i,j}$ represents the j^{th} label of the i^{th} disjunctive label set.

Conjunctive Normal Form (CNF) is the label set form that is needed at the algorithm level, so the DNF form must be converted, by means of standard Boolean algebra. The example above can be rewritten as:

(L_1 AND $L_{2,1}$ AND $L_{3,1}$ AND …) OR (L_1 AND $L_{2,2}$ AND $L_{3,2}$ AND …) OR …

In this CNF expression, each parenthetical statement is a Conjunctive Label Set (CLS). In the KPM, each CLS is used in the generation of a wrapping key. Each label in a CLS is used to calculate a shared value, and the shared values become the input to a key derivation function that outputs a CLS-specific wrapping key.

3.5.2 Special Labels

Special labels called sensitivity labels are used to specify how data is handled. These labels denote the type and strength of the identification and authentication needed to log in, the minimum strength of cryptographic algorithms that can be used, whether a digital signature is required, and what integrity and size the digital signature should be. For an encryption operation, exactly one sensitivity label is used. In the previous disjunctive and conjunctive example equations, the $L_{1,1}$ term represents the sensitivity label.

Another special label is the foreign label, which is a label from another domain. In the domestic domain, a corresponding shadow label is created. Each domain has its own set of parameters, so each label has different domain parameters associated with it. Whenever the foreign label is used in an operation, the corresponding shadow label must also be used in order to allow key recovery within the domestic domain.

3.6 KPM Key Wrapping

The labels from the previous section are used to generate wrapping keys unique to each CLS. Since labels consist of public/private key pairs, the public keys can be used to encrypt data in such a way that only members with the corresponding private keys have access to the data. This is done by generating a number of CLS-specific wrapping keys to protect one symmetric encryption key.

3.6.1 Shared Values

Shared values, SV_j, are generated for each label in the CLS. For encryption, the shared values are calculated by multiplying the private ephemeral key (d_e) with the label's public key ($\boldsymbol{Q_j}$) and taking the x-value of the resulting point. The process is

$$\boldsymbol{Z_j} = d_e\boldsymbol{Q}_j$$

$$SV_j = x_{Zj}$$

where j is the index of the j^{th} label. After encryption, these shared values are discarded, and the shared values for one CLS need to be generated again during the decryption. This entails multiplying the public ephemeral key ($\boldsymbol{Q_e}$) and the private label key (d_j), as follows:

$$\boldsymbol{Z_j} = \boldsymbol{Q_e}d_j$$

$$SV_j = x_{Zj}$$

3.6.2 Key Encryption Key

The key encryption key is derived by means of the key derivation function of ANSI X9.63, Section 5.6.3 (Accredited Standards Committee ANS X9.63, 2002):

$$KEK = kdf(Z, SharedInfo)$$

Z is a concatenation of the shared values:

$$Z = x_{Z1}||x_{Z2}||\ldots||x_{Zd}$$

where d, is the number of labels in the CLS. The *SharedInfo* parameter is

$$SharedInfo = L_1||L_2||\ldots||L_d$$

where L_j refers to the GUID of the j^{th} label. The key derivation function makes use of the SHA-512 hash function (National Institute of Standards and Technology FIPS Pub 180-2, 2002). Note that the size of the key encryption key is 256 bits.

3.6.3 Key Wrapping

The Advanced Encryption Standard (AES) key wrap function is used to wrap the data encryption key (National Institute of Standards AES Key Wrap, 2002). For encryption, the process is

$$wK_i = W_{KEKi}(K)$$

where K is the data encryption key, W_{KEKi} is the key wrap function, KEK_i is the key encryption key from the i^{th} CLS, and wK_i is the wrapped key. Similarly, for unwrapping, the process is

$$K = U_{KEKi}(wK_i)$$

where U_{KEKi} is the unwrap function.

3.7 KPM Symmetric Encryption

The symmetric encryption used in the KPM is the 256-bit key AES algorithm in cipher-block chaining (CBC) mode (National Institute of Standards AES, 2001). For encryption, the process is

$$C_i = E_K(P_i \text{ XOR } C_{i-1}),\ C_0 = IV$$

where P_i is the i^{th} block (of 128 bits), E_K is encryption operation with key K, IV is the initialization vector, and C_i is the i^{th} encrypted block. For decryption, the process is

$$P_i = D_K(C_i) \text{ XOR } C_{i-1},\ C_0 = IV$$

where D_K is the decryption operation with key K.

3.8 N2K Packet Header

The N2K packet header accompanies the encrypted data. Its purpose is to package the metadata needed to manage packets successfully, such as the author's user ID, wrapped keys, digital signatures, and the ephemeral public key. The packet also carries any additional information needed to recreate the original data. The packet contents can be seen in Table 1. In the table, c is the total number of CLSs, d is the total number of disjunctive label sets, g is the total number of domains spanned by the chosen labels and u is the number of authoring users.

Table 1: The N2K packet header contains metadata needed to successfully decrypt the message (InfoAssure N2K, 2004).

Name	Symbol	Type	Size	Instances
Version	v	Short Integer	2	1
DateTime	DT	Character	14	1
Wrapped Key	wK_i	Value	8 – 32	c
Initialization Vector	IV	Value	8 – 16	1
Algorithm	A	Code	4	1
Ephemeral Public Key	Q_j	EC Point	286	g
Domain ID	D_j	GUID	16	g
Label ID	L_l	GUID	16	$c \cdot d$
Label Maintenance Level	M_{L_l}	Short Integer	2	$c \cdot d$
Inner Digital Signature	S	Value	256 – 512	0 or more
Digital Certificates	X	X.509	500 – 2000	0 or more
User ID (if no certificate)	uid	GUID	16	u
Outer Digital Signature	SH	Value	256 – 512	1
Other Data	O	Binary	Variable	0 or more

4 Implementation

This implementation makes use of the standard simulation-based FPGA design flow, shown in Figure 14. First, the design requirements are enumerated. Next, an architectural specification is developed with register transfer logic (RTL); its behavior is specified using a hardware description language such as VHDL. Then, functional simulations are performed until the desired level of correctness is reached. Following correct behavioral simulation, the flow is passed through a synthesis tool to translate the description into generic logic primitives. Simulations are performed to check the design after synthesis. The next step is placing and routing of the design logic, which is accomplished by vendor-specific tools. A third simulation step can be used to verify the design with all of its logic and routing delays. Finally, the implementation is programmed into the FPGA.

4.1 KPM Architecture

The KPM architecture is that of a tightly-integrated cryptographic coprocessor; it is external to a host CPU, and it performs all of the operations needed to implement the KPM algorithm. The host CPU gives high-level commands to the coprocessor and supplies the necessary N2K packet data. The CPU and coprocessor are linked by a bus with 32 or 64-bit data width, and communication is via shared dual-port RAM. This arrangement is shown in Figure 15.

The coprocessor's main functional block is the controller, which handles the high-level steps of the KPM algorithm. The controller directs the shared memory operations and also controls the various cryptographic modules, which implement specific cryptographic algorithms. To optimize the area of the design, each module is only used once within the design; the controller arbitrates access to the cryptographic modules since more than one KPM step might require the same module. This organization is shown in Figure 16.

Rather than using a bus, direct links are used between the controller and the cryptographic modules. A bus would add latency since it would require large

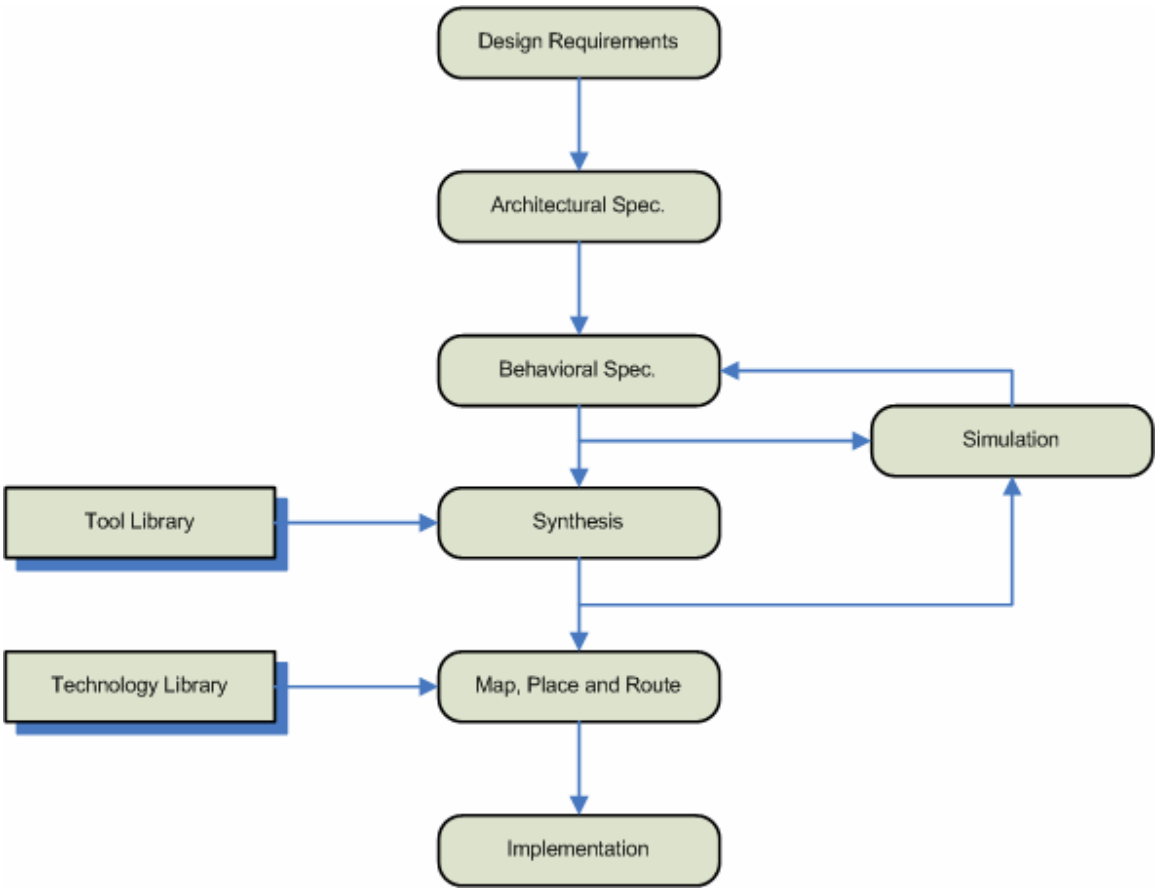

Figure 14: Simulation-based FPGA involves multiple simulation cycles before the final implementation.

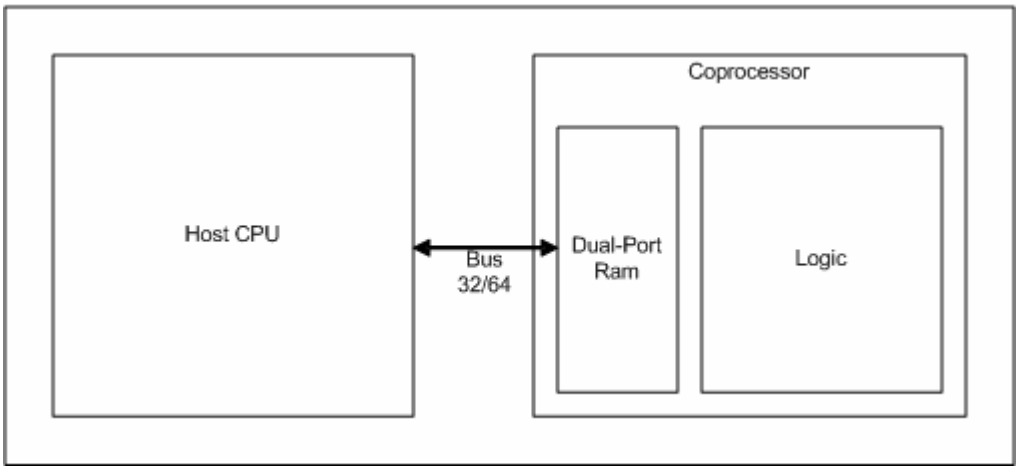

Figure 15: The KPM architecture is a tightly-integrated coprocessor external to the CPU.

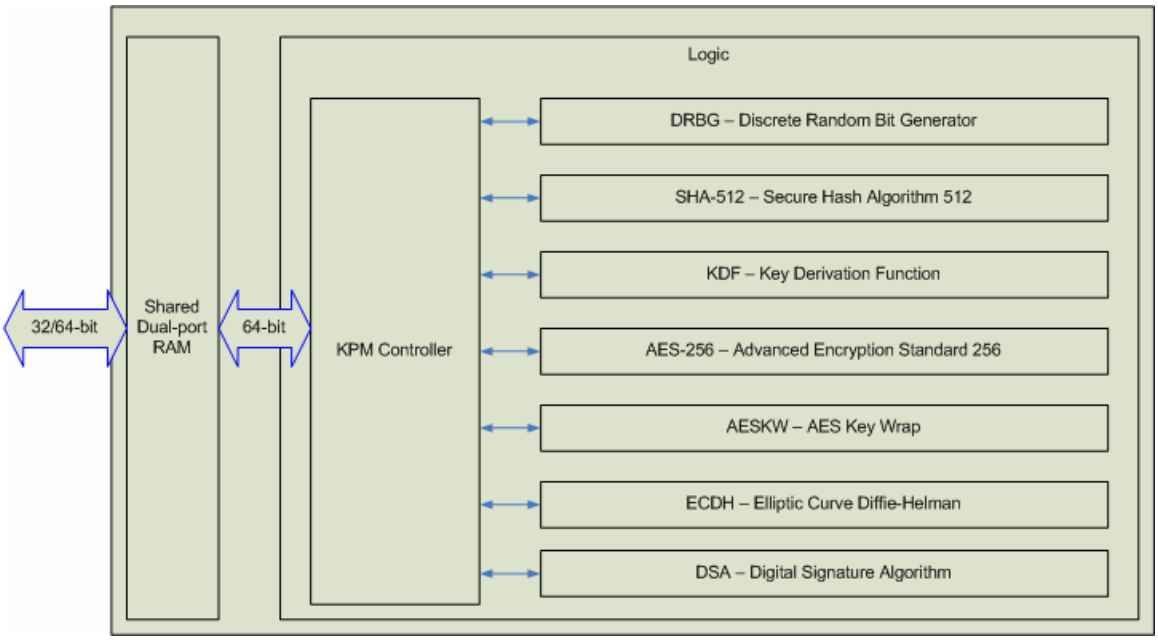

Figure 16: The KPM architecture is a controller directly linked to many cryptographic modules.

cryptographic words to be split into bus-width vectors and sent over multiple clock cycles. While this might be less of a problem if the cryptographic modules were all pipelined, many of them are not, in the interest of saving area. Using direct links, which offer decreased latency, has the downside of increasing routing complexity.

4.1.1 Memory Map

The shared dual-port is divided into four regions based on function, as depicted in Figure 17. The first and second regions are for control to the KPM and to the host CPU, respectively. The third region holds the N2K metadata and CPU-generated data, such as the random seeds, elliptic curve parameters, and signing keys, and label sets. The fourth region holds the message cache.

Region 1 bits set the configuration of the current operation, and a single 64-bit word is used for this purpose. Using this word, the CPU selects whether the coprocessor will proceed with encryption, decryption, or built-in self test (BIST). It also sets other information not included in the N2K header, such as the status of the message cache and

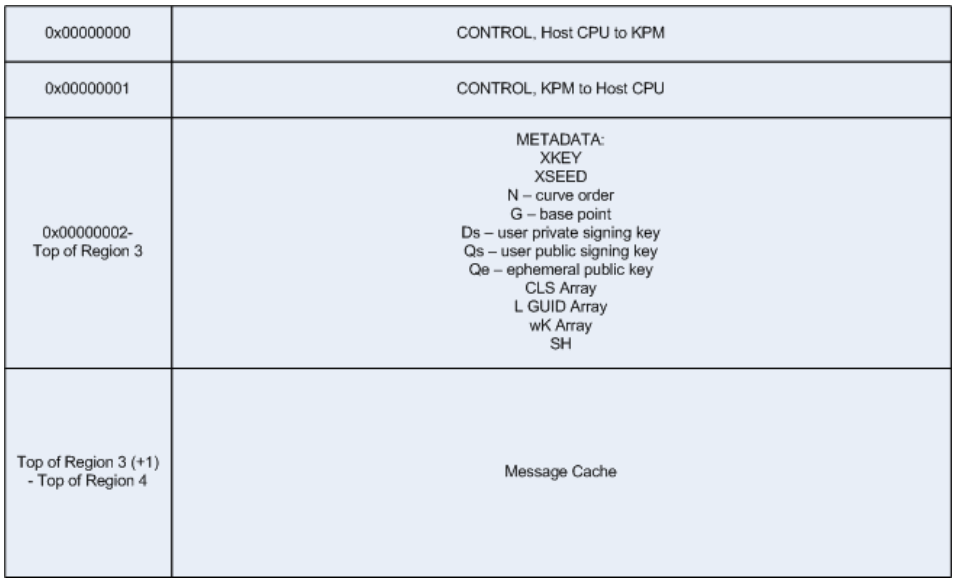

Figure 17: The memory map defines four regions: two for control, one for metadata, and one for the message cache.

whether to use the default curve parameters (a testing feature). The significance of each bit in region one is given in Table 2.

Region 2 bits report the coprocessor status to the host CPU, and a single 64-bit word is reserved for this region. The 0-byte is used for mid-operation status, namely whether the cache has been replenished with encrypted data and is ready to be read by the host CPU. The 1-byte is used for end-of-operation status that reports bad metadata keys or signatures. The remaining used bits are used to report BIST status and to specify the exact cause of failure. The used bits and their descriptions are shown in Table 3.

Region 3 contains the N2K metadata needed for the desired operation, as listed in Figure 17. It also contains large vectors of data generated by the CPU, such as true-random seed values. In the implementation, each memory location is defined relatively so that the physical memory locations are assigned at compile time. This facilitates future updates to the memory map, releasing the designer from the need to manually recalculate how each metadata field should fit into the 64-bit memory slots.

The fourth region, or memory cache, is used to hold subsets of the message data. The host CPU initially writes input data to the cache, and the coprocessor overwrites this input with the processed output. The existence of a memory buffer such as this cache

Table 2: Region 1 contains the Host CPU to KPM control bits.

Bit	Name	Description
0	ENC_DEC_BIT	1=encrypt, 0=decrypt
1	BIST_BIT	1=run BIST, 0 = normal
2	LAST_CACHE_BIT	1=last cache fill/partial, 0=more cache fills ahead
3	B571_BIT	1=B571 curve, 0=K571 curve
4	DEFAULT_G_BIT	1=use default curve base point, 0=specify point
5-7	Unused	
8-15	CACHE_SZ	Number-1 of 128-bit message blocks in cache
16-23	C	Number of CLS blocks in cache
24-63	Unused	

Table 3: Region 2 contains the KPM to Host CPU control bits.

Bit	Name	Description
0	CACHE_FULL_BIT	1=done with cache, 0=ready or processing
2-7	Unused	0-byte reserved for mid-operation status
8	DONE_BIT	1=done with everything, 0=processing caches
9	Qe_INVALID_BIT	1=invalid ephemeral public key (dec)
10	WK_INVALID_BIT	1=invalid wrapped key (dec)
11	OS_INVALID_BIT	1=invalid outer signature (dec)
12-15	Unused	1-byte reserved for end-of-operation status
16	BIST_ENC_CTRL_FAIL_BIT	1=failed enc ctrl BIST
17	BIST_DEC_CTRL_FAIL_BIT	1=failed dec ctrl BIST
18	BIST_ENC_FAIL_BIT	1=failed enc encrypt BIST
19	BIST_DEC_FAIL_BIT	1=failed dec decrypt BIST
20	BIST_WK_FAIL_BIT	1=failed key wrap generation BIST
21	BIST_OS_FAIL_BIT	1=failed outer signature generation BIST
22-63	Unused	

allows burst transfers that can increase performance; the host CPU can fill the cache and then perform some other tasks while waiting for the coprocessor to continue. Also, since the shared memory is of limited size, it is possible that multiple transfers of data will be required in order to fully process a single encryption request. It is assumed that the memory is large enough to contain all of the metadata but that it can only accommodate a fraction of the message; that fraction is loaded into the message cache. The host CPU and the coprocessor may need to read and update their control signals several times before the full message is processed.

4.1.2 Main Controller

The main controller implements a state machine to oversee all operations in the coprocessor. The controller sits in ready state until receiving a start signal. It then fetches the control word from memory, decodes the configuration bits to determine the operation, and proceeds with the desired operation. For encryption and decryption, sub-state machines are used. For the built-in self test, load and check states are bound to both the encryption and decryption states. Upon completion of the operations, the controller returns to the ready state. The main state machine is depicted in Figure 18.

4.1.3 Encryption

The encryption state machine follows the KPM algorithm through all the states necessary for encryption. From the initial ready state, the curve parameters are loaded and the pseudo-random key (*K*), initialization vector (*IV*), and ephemeral private key (d_e) are generated. Then, the ephemeral public key (Q_e) is calculated. Loops are used to find the shared values (*SV*) from the CLSs and CLS GUIDs, to generate the key encryption keys (*kek*), and to generated the wrapped keys (*wK*). Following these operations, the encrypted data and hash are generated. Finally, the inner and outer signatures are computed. The entire cycle is depicted in Figure 19.

Exploiting parallelism at the KPM level is in some cases possible, when both (a) successive states do not require the same module and (b) the function inputs have been previously generated. Steps 8-10 of the KPM algorithm meet this requirement. Since

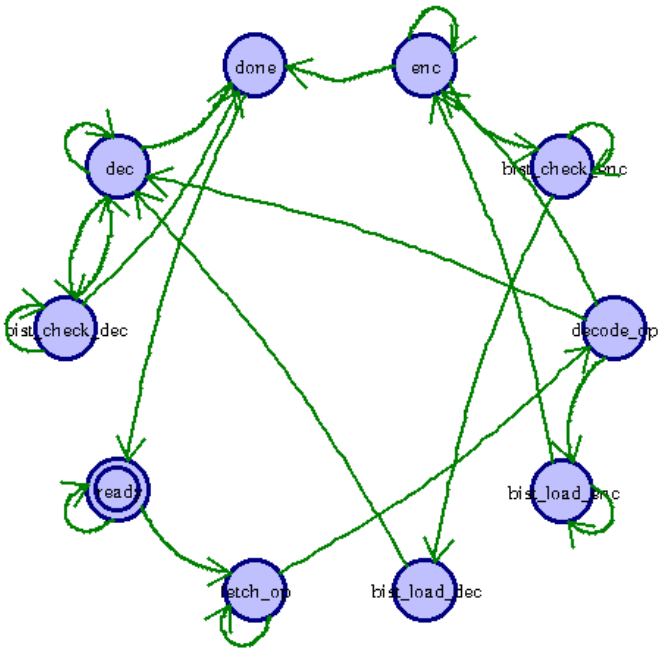

Figure 18: The main state machine oversees KPM operations.

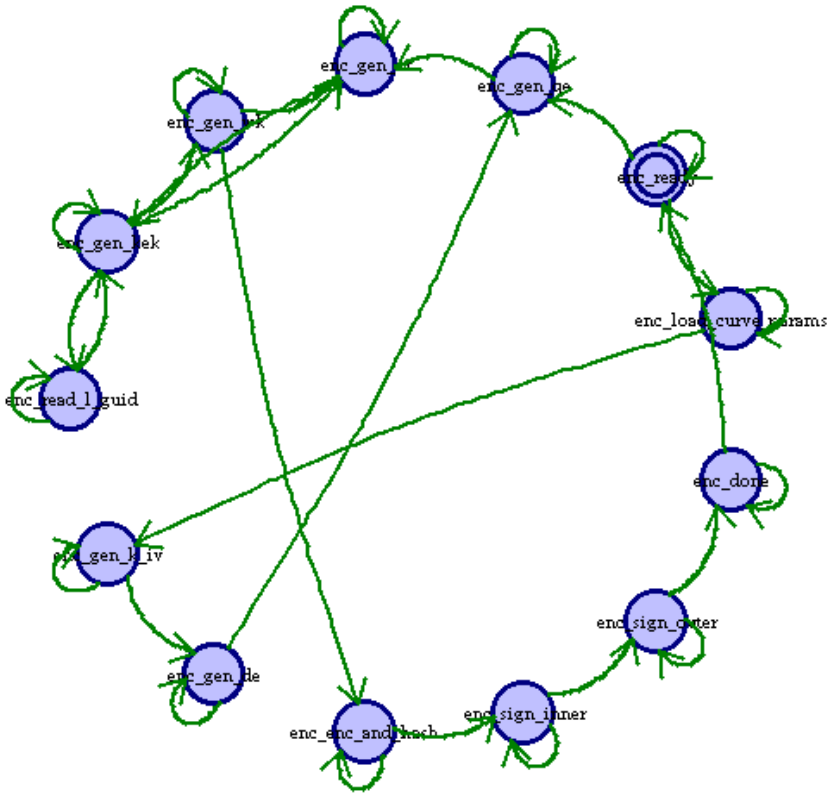

Figure 19: The encryption state machine follows the KPM algorithm.

these steps encrypt and sign the message which can theoretically range from 0 to 2^{128} bits, they stand to be a major bottleneck in the KPM. Additionally, they encompass the data passing with the host CPU, which can suffer from high latency. Thus, these relatively slow operations are parallelized as follows: message blocks are read from shared dual-port RAM, the data is encrypted (using AES-256), the N2K package is constructed in RAM, the hash value is generated (using SHA-512), and requests for more data are sent to the host CPU. The cycle graph is shown in Figure 20. The scheme needs to be unusually flexible since the AES-256 module is multi-cycle, the RAM is pipelined, and the SHA-512 module is both pipelined and sometimes multi-cycle.

The effect of this parallelism can be examined using Amdahl's Law:

$$Speedup = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

which sets a bound on the performance gain (Hennessy and Patterson, 2003). The $Fraction_{enhanced}$ term is the portion of the cycles used by SHA-512, and $Speedup_{enhanced}$ is the speedup of SHA-512 due to parallel computations. Using a 17-cycle encryption module and 80-cycle SHA-512 module, the SHA-512 is responsible for 33% of the cycles and 25% of its cycles are in parallel. Thus, the speedup is a modest 1.09. Further enhancement of this block would require additional modules or modules with lower-latencies.

When multiple recipient roles are specified, steps 4 and 5 also meet the parallelization requirements. Making use of a two-stage pipeline, step 4 could begin generating a new KEK while step 5 wraps K with the previously generated KEK. The implementation, however, does not take advantage of this. While steps 8-10 are potential bottlenecks, steps 4 and 5 generally consist of relatively few hashes and encrypt operations, so no effort is made to pipeline them.

4.1.4 Decryption

The decryption state machine follows the KPM algorithm through all the states necessary for decryption. From the initial ready state, the curve parameters are loaded and validated. The shared value (*SV*) from the appropriate CLS and CLS GUIDs is

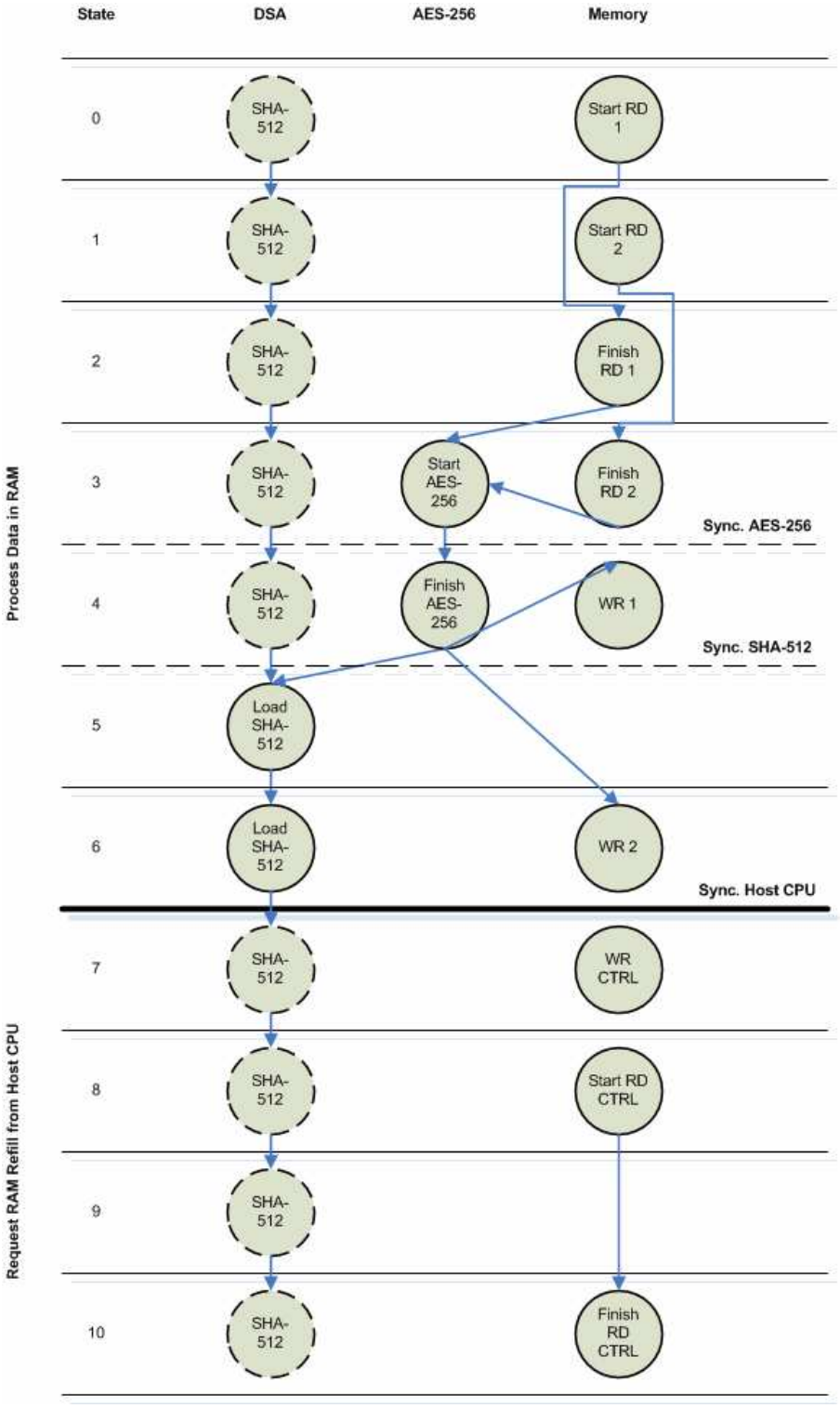

Figure 20: Parallelism is exploited during steps 8-10 of the KPM algorithm.

found, and one of the key encryption keys (*kek*) can be calculated. Loops are needed to traverse the memory arrays of CLS and CLS GUIDs. With the *kek*, a wrapped key (*wK*) can be unwrapped to reveal the data encryption key (*K*). The message can then be decrypted, and a hash can be generated. After the entire message has been passed through the message cache, the outer and inner signatures can be verified. The entire cycle is depicted in Figure 21.

Parallelization potential for the decrypt operation is very similar to that for encryption. The SHA-512 and AES-256 modules can be run in parallel as the message is being processed. The main difference between the encryption and decryption operations is the order of operations – in the decrypt case, the signature is generated with the input to the AES-256 module rather than with its output.

4.1.5 Built-In Self Test (BIST)

The BIST mode simulates both encrypt and decrypt requests from the host CPU. Initially, it loads test data into dual-port RAM. Then it begins the encryption process. Upon completion, the results in RAM are compared to internally-stored "golden" outputs, and the discrepancies are noted. Control signals, generated keys, encrypted results, and hash results are tested in this manner, and the state machine for these checks is shown in Figure 22. The memory-loading process is then repeated to test decryption, and the decrypt BIST checks the control signals and decryption results. The decrypt BIST can be simpler since a number of verification steps are built into the decryption steps themselves. When the BIST completes, it returns the status of its various modules to the host CPU.

BIST tests are based on standard test vectors for the individual cryptographic algorithms. Currently, tests use four basic vectors along with keying materials from the AES Key Wrap specification. The golden vectors are stored in parameterized arrays stored in look-up tables. Increasing the number of test vectors is a matter of changing the array size and updating the array values, and if the test vectors are increased to a sufficiently large size then the synthesis tool will make use of on-FPGA RAM resources.

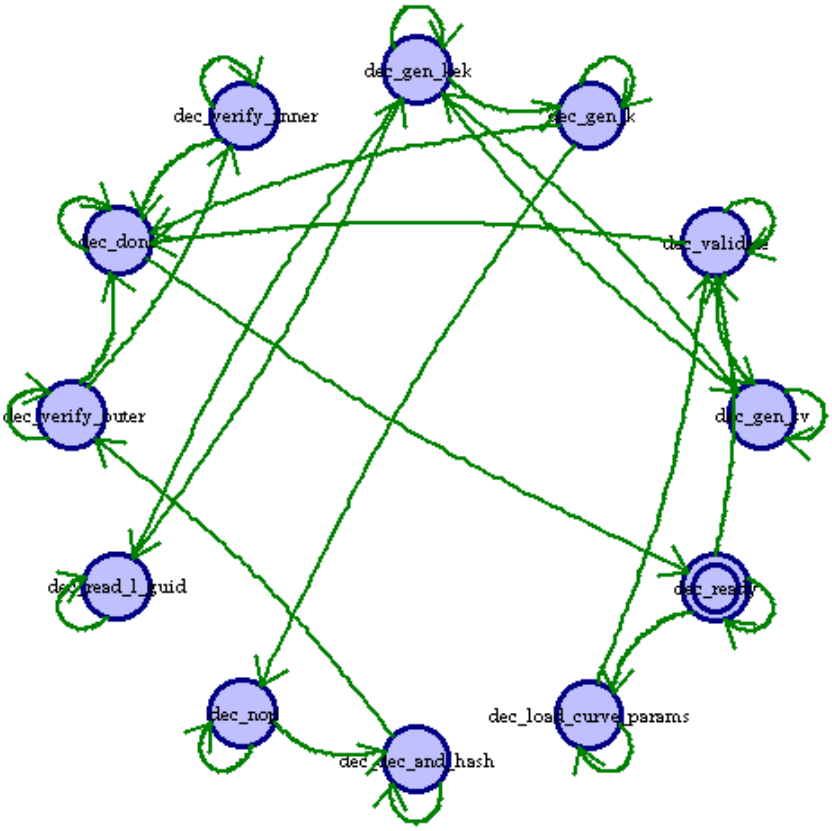

Figure 21: The decryption state machine follows the KPM decrypt algorithm.

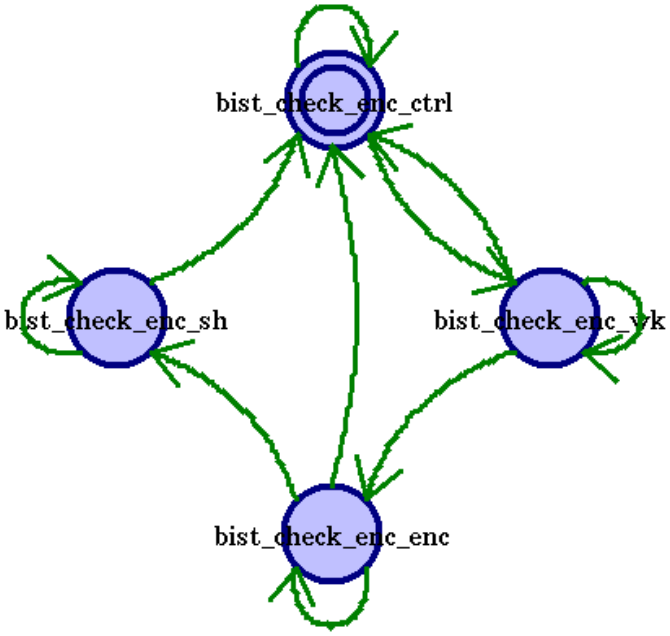

Figure 22: For encryption, the BIST checks for wrapped key, encryption, and hash results.

4.2 Cryptographic Modules

While the main contribution of this work is the KPM controller, the controller is necessarily dependent upon the design of its slave modules. To implement and test the controller, the modules had to be created and tested themselves. While the modules are not the focus, their implementations are briefly discussed below, as their design affects the design and performance of the entire system.

4.2.1 Advanced Encryption Standard (AES-256)

The AES-256 algorithm performs symmetric encryption operations for the KPM. The algorithm consists mainly of substitution, addition, and XOR operations, making it relatively efficient for hardware implementations. There are a number of tradeoffs when choosing an implementation for AES-256, and the chosen algorithm aims for relatively low area usage with modest performance. The implementation uses a non-pipelined, LUT-based loop architecture. It uses 17 cycles for encryption or decryption, and there is

an additional penalty for the key scheduler whenever a new key is loaded for decryption. While the LUT-based approach uses slightly more logic than a RAM-based approach, it offers improved performance. Performance is lessened by the choice of a non-pipelined and looped architecture, but this configuration offers significant area savings.

4.2.2 AES Key Wrap (AESKW)

The AES Key Wrap algorithm packages an encryption key by encrypting it with a data integrity check. For key wrapping, a key wrapping key is supplied, and the input key is encrypted for secure storage and transmission. Upon unwrap, the wrapped key can be recovered, and the algorithm verifies whether the correct result was generated. The algorithm involves shifts and XOR operations and is built on top of AES-256. Thus, the KPM controller passes control of the AES-256 module to the AESKW module when key wrapping/unwrapping needs to take place.

4.2.3 Deterministic Random Bit Generator (DRBG)

The DRBG implements the N2K *png* function by means of the SHA-1 algorithm. The SHA-1 algorithm is efficiently implemented in hardware since it consists of rotation, addition, and XOR operations. The core of the algorithm takes a 512-bit input block made of 32-bit words and produces 160-bit output, taking 80 cycles to process one block. In a typical SHA-1 implementation, the hashing operation is wrapped with a padding operation, but wrapping is not required for the DRBG implementation.

Random number generation begins with a true random seed passed from the host CPU. The seed is used as the initial input to the SHA-1 hash function, and an initial block is processed by SHA-1. The initial return value is stored for later use as well as being used to generate a new input to the hash function. Then, the hashing process is repeated with the new input, and the new output is stored. The process repeats until it has generated a sufficient number of pseudo-random bits.

Though the output from the hash function is a multiple of 160-bits, KPM keying material needs to be either 571-bit (for elliptic curve keys) or 256-bit (for AES keys). For EC, the value must belong to a particular field, so the modulus of a 640-bit number

needs to be found. The implementation thus makes use of a generic-divisor binary division algorithm. For the symmetric key, the process is simpler, and the 256-bits can simply be extracted from the concatenated hash outputs.

4.2.4 Digital Signature Algorithm (DSA)

Generating a digital signature via DSA involves applying elliptic curve encryption to a message hash. The SHA-512 module is used to hash the message contents. Unlike with the AESKW module's use of AES, the DSA module does not directly control the SHA-512 module. Rather, hashing is carried out under the direction of the KPM controller, in order to keep the elliptic curve operations as independent modules. The motivation for separating the elliptic curve modules is to keep them as black boxes during the testing of the KPM controller. The elliptic curve math was not implemented at the time of the controller's design, so the DSA implementation is merely a stub architecture for testing.

4.2.5 Elliptic Curve Diffie-Hellman (ECDH)

The ECDH module is designed to perform one of three functions. First, it can validate a public key input point. Second, it can generate a public key from a private key and base point. Third, it can generate the shared value result from input public and private keys under the given curve parameters. As with the DSA module, the ECDH module is not fully implemented because the elliptic curve math designs were not available at the time of the KPM controller implementation. A stub module was designed with the necessary inputs and outputs to allow the ECDH module to base its calculations on the current session's curve parameters.

4.2.6 Key Derivation Function (KDF)

The KDF module uses a hashing function to generate keying material for AESKW. It operates under a similar principle as the DRBG: the input values are concatenated and hashed. With the KDF, however, there are two distinct input arrays of non-pre-determined length: the shared values and the CLS GUIDs. First, the shared

values are concatenated, then a count of the keys to be generated is added, and finally the GUIDs are appended.

When the KDF module is in use, the KPM controller relinquishes complete control of the SHA-512 module. The input concatenation is built dynamically and streamed to the SHA-512 module. Once hashing is complete, a 512-bit hash is output. From this result, 256 bits are saved and used as the wrapping key.

4.2.7 Secure Hash Algorithm (SHA-512)

SHA-512 is the hash algorithm that is used for generating digital signatures. As discussed previously, the SHA-512 algorithm is expected to be part of the bottleneck in the KPM implementation. Unlike the SHA-1 implementation, the SHA-512 algorithm is pipelined to begin processing before the full input block has been passed. Inputs are 1024 bits wide and composed of 64-bit words, and the core processing takes 80 cycles. Post-processing can add another 80 or 160 cycles.

Except for the larger input block and data words, the SHA-512 algorithm is very similar to SHA-1 with minor arithmetic differences. Whereas the SHA-1 is encapsulated by the DRBG algorithm, the SHA-512 algorithm is manipulated directly by the KPM controller. Because of this, the block wrapping function is also implemented. The block wrapping functionality pads input blocks to the correct size and, if necessary, adds an empty block on the end of the input stream.

5 Results and Discussion

The design was built incrementally, and each module was tested after it was specified. Then, upon adding one module to the controller and updating the control logic, the integrated whole was verified before repeating the process for another module. This iterative testing process was used to verify and debug the RTL design completely. Following the incremental verification, the design was synthesized into a netlist. The synthesized results were then re-verified before being passed through placement and routing. In the following sections, the results of these processes are presented.

5.1 RTL Verification

Two types of verification were used to test the KPM controller design. The Mentor ModelSim tool was used for functional (and timing) simulation, and the Cadence FormalCheck tool was used for formal verification. While functional verification verifies the basic functionality of the design, formal verification checks for the existence of hard-to-locate deadlocks, control failures, and corner-case bugs. Functional verification is standard practice for any logic design, and it was used on both the modules and the controller. Formal verification is most applicable to control-oriented designs, making the controller a good candidate since it offers significant complexity in the steps related to parallel encryption, hash computations, memory accesses, and host CPU communication.

5.1.1 Mentor ModelSim

As part of the incremental testing, initial test benches were applied to verify the behavior of individual modules. In general, a test bench was built to test each of the modules using standard test vectors. The test benches made use of assertions, which allowed the correct functionality to be specified within the test bench. Using this approach, it was possible to run iterative tests from the console, letting the test bench automatically verify outputs. An example of successful output for the console-based testing of the AESKW module is shown in Figure 23. Of course, the graphical user interface (GUI) can also be used to gain a visual understanding of the module function, as

```
# ** Note: ----------  BEGINNING TESTBENCH  ----------
#    Time: 2 ns  Iteration: 0  Instance: /aeskw_tb
# ** Failure: ------  TESTBENCH COMPLETED SUCCESSFULLY  ------
#    Time: 70575 ns  Iteration: 1  Process: /aeskw_tb/monitor File: ../vhdl/tb/aeskw/aeskw_tb.vhd
# Break at ../vhdl/tb/aeskw/aeskw_tb.vhd line 191
```

Figure 23: Assertion-based test benches facilitate automatic iterative testing of modules.

shown in Figure 24. Using the GUI can be helpful for debugging but can be a hindrance to iterative testing. Each of the implemented modules was successfully tested using assertion-based functional test benches.

For integrated testing of the controller combined with modules, the controller test bench was written. Since the KPM controller includes the BIST, the controller test bench was designed to take advantage of this. When a new module was added to the controller, the BIST would also be updated. Then, to test the new system, the test bench would initialize the BIST. The new BIST logic then tested the new module code, and the test bench monitored the BIST return values for success or failure. This technique eased the task of writing the test bench; since the test bench was able to test functionality at a high level, it was only necessary for it to check a small number of return values, as shown in Figure 25 and Figure 26.

As the KPM controller was built, it was often necessary to test partially-functioning versions. Empty or reduced-behavior states were often added to the state machines, and assertions were useful in these instances to monitor when these states were being entered or exited. Additionally, when the states needed to operate on a non-existent module, a stub module was inserted into the design. The stub modules define the interface for the module but do not fully define the behavior of the module.

5.1.2 Cadence FormalCheck

By applying formal verification tools such as Cadence FormalCheck, it is possible to guarantee that a design's logic behaves as desired. Several challenges exist, ensuring that formal techniques are not a verification panacea. One problem is that these techniques suffer from state-space explosion, making it computationally infeasible to

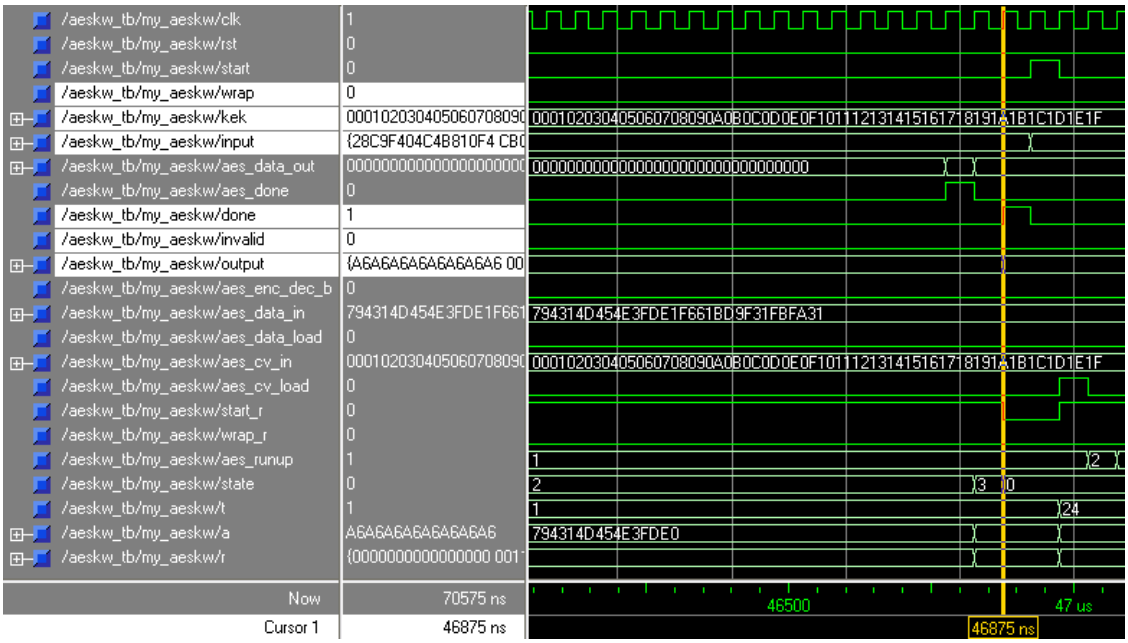

Figure 24: Functional simulation with the GUI is useful for debugging module behavior.

```
assert not din(BIST_ENC_CTRL_FAIL_BIT)='1'
report "BIST failed during CTRL (ENC)"
severity warning;

assert not din(BIST_WK_FAIL_BIT) = '1'
report "BIST failed during KEY WRAP GENERATION (ENC)"
severity warning;

assert not din(BIST_ENC_FAIL_BIT)='1'
report "BIST failed during ENCRYPT"
severity warning;

assert not din(BIST_OS_FAIL_BIT)='1'
report "BIST failed during OUTER SIGNATURE GENERATION (ENC)"
severity warning;
```

Figure 25: The top-level controller test bench only needs to check the BIST report bits.

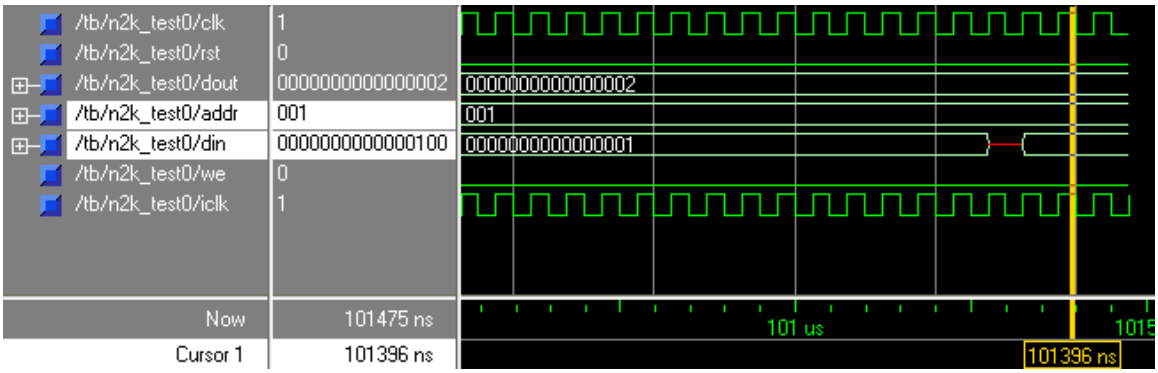

Figure 26: The controller test bench only needs to check a single memory address and return vector.

verify large designs. This problem manifests itself especially when trying to verify a design's data path. A 32 or 64-bit vector can be too much for formal verification techniques to handle. With the KPM controller, a number of 128, 256, and 571-bit vectors are used, so formal verification must be applied with care.

There are a number of ways to limit the number of states applicable to a formal verification task. One of these methods is simply partitioning the design into smaller verification tasks according the natural design hierarchy. A complimentary method is to restrict the signals. Some environmental restrictions are necessary for correct operation of the circuit, such as defining a clock and a reset. Other restrictions might reflect the behavior of logic external to the design. Furthermore, since control logic is the most likely place to find logic errors, data path calculations can be largely ignored. Large arithmetic operands and results can be considered to be short vectors for the purpose of formal verification, and then the data flow verification task is relegated completely to functional verification. All of these techniques were used in the verification of the KPM controller (Fields "Formal Verification," 2005).

FormalCheck is based on logical properties and queries. Properties specify particular design behaviors, and queries link properties that are assumed with properties to be verified. The properties rely on user-defined windows of interest, which consist of several conditions: enabling condition, fulfilling condition, and discharging condition. A window is triggered by one event and released by another, and during the window some condition is examined. Only the fulfilling condition is necessary – leaving off the

enabling condition implies that the window is initially enabled, and leaving off the discharging condition implies that the window is never discharged. A user specifies an enabling condition using *after* or *if_repeatedly*. Fulfilling conditions are specified with *always*, *never*, *eventually*, or *eventually_always*. Discharging conditions are specified with *unless* and *unless_after*. Semantically, these specifications are the same as standard English.

Properties are used to specify two types of functional requirements: safety requirements and liveness requirements. Safety requirements state that the design should never behave in certain ways, while liveness requirements state that the design should always be able to respond to certain stimuli.

One safety requirement is that the device should reach the *ready* state immediately after a reset is issued. The requirement is triggered on a rising edge during which the reset signal is *falling* and the *i_top_attn* start signal is 0. The requirement is that the top level state machine is always in the *ready* state for one cycle after the trigger event. Because of the asynchronous reset and the way the timing is specified in this property, the enabling condition must include *i_top_attn = 0* or else the state machine could advance immediately after the reset signal goes low. This is defined as:

```
property -create timely_startup {
        after {
                n2k_control:clk = rising and
                n2k_control:rst = falling and
                n2k_control:i_top_attn = 0
        }
        always {
                n2k_control:state = ready
        }
        within -delay 0 -duration 1 { n2k_control:clk = rising }
}
```

The main signaling from the KPM to the CPU is the assertion that processing has completed. Thus, another safety requirement is that the “done bit” should only be asserted when processing actually has completed. Processing has been completed when the top level state machine is “done”, and only in that state should the “done bit” *n2k_control:o_top_dout(8)* be written to *CONTROL_ADDR_OUT*. This property is:

```
property -create only_done_bit_when_done {
        never {
                n2k_control:o_top_we = 1 and
                @CONTROL_ADDR_OUT and
                n2k_control:o_top_dout(8) = 1
        }
        unless { n2k_control:state = done }
}
```

Since the control words to and from the host CPU are mutually exclusive, the KPM controller should never write to the CPU's word. This is a third safety requirement, stated simply:

```
property -create never_write_cpu_word {
        never {
                n2k_control:o_top_we = 1 and
                @CONTROL_ADDR_IN
        }
}
```

Liveness requirements are used to ensure that the design cannot hang in a certain state. The CPU should always be able to issue commands such that the KPM processes input and returns a result. The return of the result is signaled by the assertion of the "done bit" in the *CONTROL_ADDR_OUT* memory location. This property is checked by:

```
property -create eventual_done_bit {
        eventually {
                n2k_control:o_top_we = 1 and
                @CONTROL_ADDR_OUT and
                n2k_control:o_top_dout(8) = 1
        }
}
```

The *eventual_done_bit* property could be fulfilled if the design were to hang in the *done* state, continually asserting the "done_bit". One way to be assured that this is not the case is to define a second liveness property, stating that the controller can always reach the *ready* state:

```
property -create eventual_ready_state {
        eventually { n2k_control:state = ready }
```

}

Of course, if the reset were asserted, the controller would revert to the ready state. However, reset was defined to be constant after an initial pulse, so together the *eventual_done_bit* and *eventual_ready_state* requirements ensure that the controller will always be responsive to valid input.

These safety and liveness properties were incorporated into queries and fully verified using Cadence FormalCheck. Thus, the design is guaranteed to perform in accordance with the properties. From the safety property verifications, it will never stall at reset, it will only signal the done bit when it has finished its processing, and it will never write into the host CPU's restricted memory space. From the liveness properties, it will always eventually notify the CPU of its completion, and it will always eventually be ready to process another requests for data.

5.2 Synthesis Results

The implementation was synthesized with Synplicity Synplify Pro targeting a Xilinx Virtex-II Pro XC2VP30 FPGA. This FPGA contains 13,696 slices that each contain two storage elements and two function generators. The device can be configured to offer up to 27,392 register bits and 27,392 look-up tables (LUTs) (Xilinx, 2005). The resource usage is broken down by modules and shown in Table 4. Note that the reported usage percentages were calculated by Synplify Pro and that hand-calculated values based on component usage and the Xilinx data sheet differ slightly.

The table shows the expected result that most of the sequential functionality takes place in the controller. The controller has high register requirements because it is buffering long cryptographic vectors as they are read from memory or passed among the various modules. This is expected since the design uses point-to-point connections to increase performance. Using a bussed architecture would reduce the register requirements but result in a negative impact on performance.

Table 4: The KPM resource usage reported by synthesis for XC2VP30 is shown.

	Register bits	**Register Usage**	**LUTs**	**LUT Usage**
Controller	15,888	53%	12,717	43%
AES-256	1,047	2.05%	3,225	6.3%
AESKW	1,042	2.04%	750	1.47%
DRBG	4,092	8.00%	16,556	32.35%
SHA-512	2,212	16.4%	5,397	10.55%
Total	*17,429*	*59%*	*39,315*	*133%*

The LUT usage of the controller, while not surpassing that of the combined modules, is also high. The chief cause for this usage is the implementation of a number of shift registers to transfer data between the wide cryptographic buffers and the narrow memory bus. This is an area-inefficient scheme whose only positive benefit is ease of specification, and it would be more efficient to implement only a single wide shift register to handle the needs of the many cryptographic buffers. In addition to the repeated shift registers, though, the large number of checks required by the state machines contributes significantly to the LUT usage.

The DRBG module stands out among the modules, since it uses nearly a third of the device's LUTs. This is because of the very large modulus operation that is required by the DRBG block. The binary division consumes a large amount of resources since it cannot be efficiently implemented in logic gates. The division is also responsible for the relatively high register usage, since very wide registers are used to shift the intermediate division values.

Second to the DRBG module, the SHA-512 module uses more resources than the other cryptographic modules. Unlike the modules with lower resource usages, this module operates on 1024-bit vectors. These operations, though implemented efficiently, result in the high LUT usage. The high register usage is a result of the buffering and padding operations needed to prepare input data for the core processing functions.

The estimated performance of the controller by itself is 68.4 MHz. The critical path here is a result of passing through hierarchical state machines and then through a

wide shift register. Once the implemented modules are added, then the estimated performance is 35.3 MHz, limited by the AES-256 module. The AES-256 design has been prototyped on an older technology with 33 MHz performance, so this performance is in-line with what is expected when using the AES-256 module with Virtex-II technology. The expected throughput, based on the AES-256 critical path and 28.6 cycles per 128-bit word, is 150Mb/s.

5.3 Post-synthesis Simulation

In accordance with the FPGA design flow in Figure 14, the synthesis result was again simulated. This post-synthesis simulation verifies that the synthesis tool correctly translated the design into a netlist, and it also includes logic delays, specified in the vendor's component libraries. Figure 27 and Figure 28 show the assertion-based test result and the BIST state transitions, respectively. These outputs show that the post-synthesis result operates correctly.

For simulation, the synthesis tool outputs a structural Verilog design description. The existence of Verilog post-synthesis components and VHDL test bench and library components can be seen in both of the post-synthesis figures – Verilog components are represented by light blue ball icons, while VHDL components are represented by dark blue boxes. Figure 27 shows the various modules, but the DSA module is missing because its stub architecture was optimized away by the synthesis tool.

5.4 Place and Route Results

Without the full implementation of the DSA and ECDH blocks, a full prototype cannot be tested. However, the existing design can be run through the FPGA vendor place and route tools, resulting in a layout that is useful for visualization. The design was fit into a Xilinx Virtex-II Pro XC2VP70 device, and the result is shown in Figure 29. The design as routed assumes that the internal block rams of the FPGA are used as the shared memory between the host CPU and the coprocessor. All Virtex-II Pro devices include embedded PowerPC cores that are suitable for use as the host CPU. However,

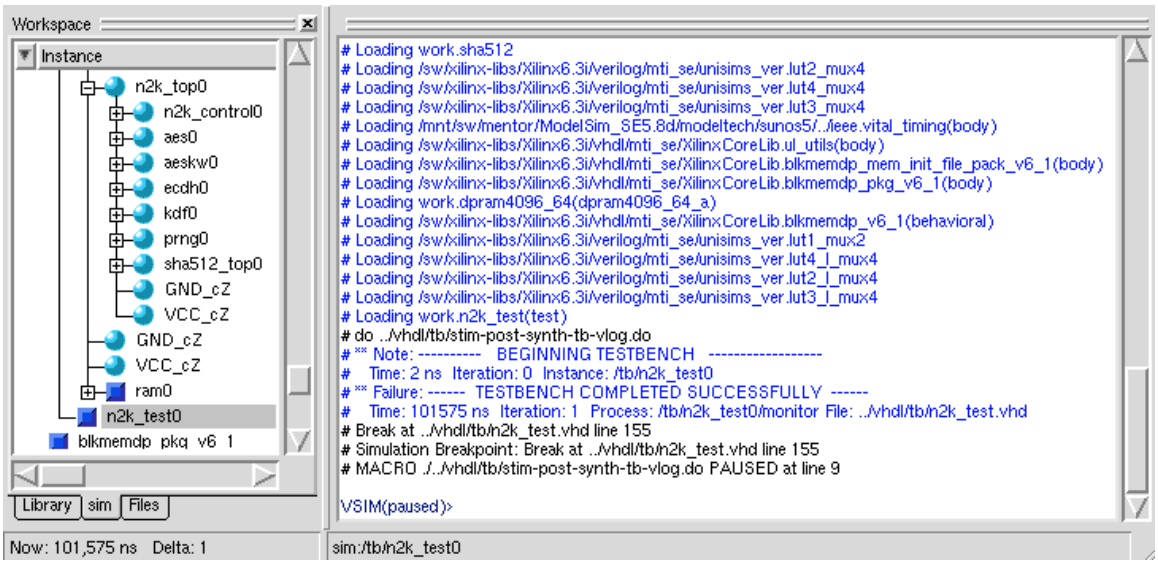

Figure 27: The assertion-based post-synthesis test bench completes successfully.

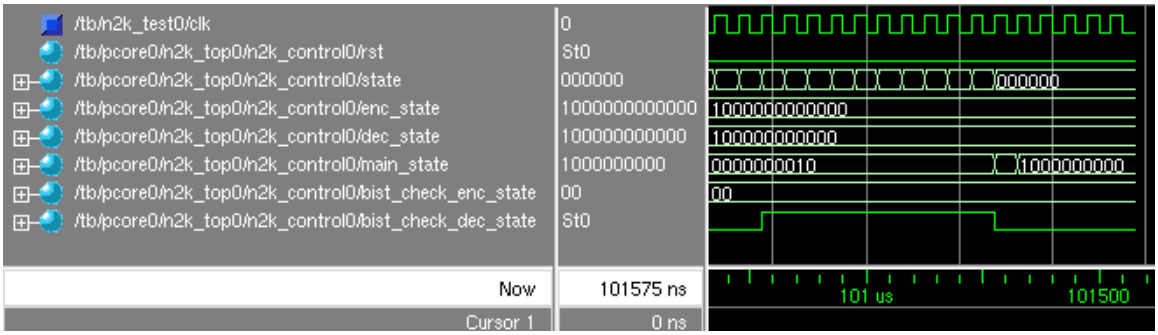

Figure 28: The post-synthesis test bench waveform corroborates the assertion-based result.

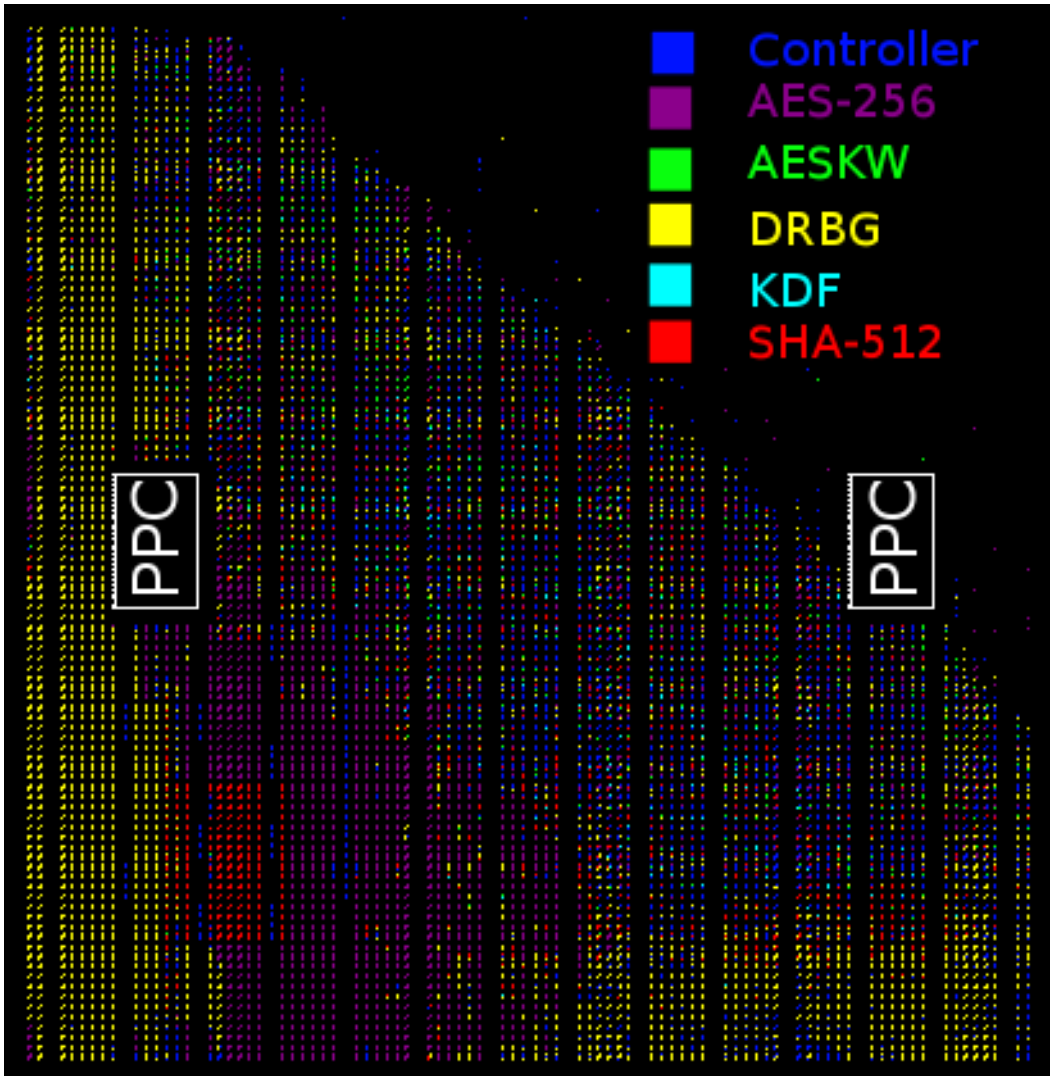

Figure 29: The result of automatic placement shows the relative distribution of logic among the modules.

this requires the use of additional glue logic, perhaps an on-chip bus, and that logic is not included in the placed and routed design shown.

The placed and routed result is colorized to reflect the area usage of the various design components, and these results mirror those of the synthesis tool. The controller takes up the majority of the chip, and the DRBG, AES-256, and SHA-512 are the largest cryptographic modules. The KDF and AESKW modules are minimal, and their logic is mixed with the controller logic. The DSA and ECDH modules are not highlighted since their stubs contribute negligibly to the resource usage.

6 Conclusion

This work leads the way to the first hardware-based implementation of the KPM. Its development is a necessary step toward highly secure and flexible deployments of the N2K system. This implementation provides the flexibility and ease-of-prototyping needed for prototypes to be developed and tested in user environments, and following these steps next-generation secure communications can be realized.

6.1 Summary

There is a clear need for improved computer security in the form of cryptographically-enforced role-based access control. The Need2Know system aims to provide this increased security and enable secure communications for government, corporate, and personal entities. Commensurate with these goals, an implementation of the N2K KPM must be designed with consideration of a number of tradeoffs: low time-to-market, high throughput, low power, low cost, feature flexibility, and implementation security. Considering the merits of various approaches, an FPGA implementation was selected.

A tightly-integrated coprocessor architecture was chosen for the FPGA implementation. This architecture implements all of the algorithms required by the top-level KPM algorithm and includes a hardware controller, so the host CPU needs only to communicate at a high level of abstraction with the controller. Various cryptographic algorithms are implemented as modules, and the modules have direct links to the controller. The controller directs their actions and is in turn controlled by the host CPU by means of shared dual-port RAM. To minimize the area requirements, each module is instantiated only once, and the KPM controller is responsible for switching control of the modules when required. The implementation includes encryption, decryption, and BIST modes of operation. The BIST includes standard test vectors to verify the operation of the cryptographic modules and the overall function of the controller.

Verification was performed iteratively throughout the design process. The cryptographic modules were functionally verified individually before being integrated

into the controller; the controller was both functionally verified with the modules and formally verified on its own. The implemented modules and controller passed all the verification tests, though a full implementation of the KPM is not possible since elliptic curve arithmetic functions for the DSA and ECDH modules have yet to be implemented.

Synthesis results for the KPM controller and modules were presented, showing a breakdown of the resource usage for various components and the expected performance of the coprocessor. The device required more area than originally expected, using more than 100% of the Xilinx Virtex-II Pro XC2VP30 device. However, the performance expectations were within range of previously prototyped modules.

6.2 Future Work

Several techniques are possible to optimize the performance of the implementation. For one, the bandwidth between the host CPU and coprocessor is typically a bottleneck in system performance. While the message cache is an improvement over single-word passing, other schemes hold the potential for higher throughput. For example, tiered memory caches could allow one memory cache to be operated on by the coprocessor while the host CPU was filling the other one. Similarly, a shared FIFO buffer would allow parallelization of data passing and data processing.

Another option for improving performance involves the connections between the modules and the controller. The point-to-point connections of the modules to the controller allow the potential for maximum performance, but there might be other more efficient means of achieving good performance. A hierarchy of busses, for instance, might make sense, or simply a 128 or 256-bit bus. Aside from the additional latency, one downside of these techniques is that the pre-existing modules would need to be wrapped in order to support communication across the bus.

Still other performance gains can be realized if the area-savings requirements are relaxed. For instance, pipelining the slowest and most frequently-used module, AES-256, would increase the total system performance. Also, modules could be replicated to increase the exploitable parallelism. The CLS processing is a good candidate for parallel processing, since a number of shared values will need to be calculated from independent

CLSs. As another option, the entire data path could be pipelined, though this mainly makes sense in a multi-user or multi-connection computing environment where multiple packets are waiting for processing.

One of the stated design considerations was low-power, and FPGA devices are known be inefficient in this respect. Nevertheless, there are several techniques that can be examined and implemented in order to reduce the power consumption of an FPGA device. Multiple clock domains and/or clock gating, for instance, can help to minimize the power consumption when the coprocessor is not in use.

Finally, since the implementation is designed to secure data, it needs to include defense mechanisms against all known security attacks, both active and passive. However, design up to this point has focused mainly on performance and flexibility. Effort should also be spent on thwarting timing attacks, differential power analysis attacks, and temperature-based attacks. Perhaps the BIST can be extended to more-fully examine the coprocessor's behavior at regular intervals and to shut down the device when an undesired intrusion is detected. Also, physical security is important, and logic integration with tamper-detecting packaging should be investigated.

References

Accredited Standards Committee X9. "ANS X9.63-2001: Public Key Cryptography for the Financial Services Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography." 2002. Available from http://www.x9.org.

Adams, L., "Choosing the Right Architecture for Real-Time Signal Processing Designs," [Online] Available: http://focus.ti.com/lit/an/spra879/spra879.pdf. Accessed August 24, 2005.

Crowe, F., Daly, A., Kerins, T., Marnane, W., "Single-Chip FPGA Implementation of a Cryptographic Co-Processor," FP, 2004.

Dandalis, A., Prasanna, V., Rolim, J., "An Adaptive Cryptographic Engine for IPSec Architectures," IEEE Symposium on FPGAs for Custom Computing Machines, 2000. [Online] Available: http://ieeexplore.ieee.org/iel5/7244/19546/00903400.pdf?arnumber=903400. Accessed August 30, 2005.

Discretix, "CryptoCell Overview," [Online] Available: http://www.discretix.com/PDF/DiscretixCryptoCell.pdf. Accessed August 30, 2005.

Eberle, H., et al., "A Public-key Cryptographic Processor for RSA and ECC," ASAP 2004. [Online] Available: http://research.sun.com/projects/crypto/ASAP2004_final_v4.pdf. Accessed August 30, 2005.

Fields, S., "Formal Verification of the Need2Know Key Protection Module using Cadence FormalCheck," 2005. Available: http://web.utk.edu/~sfields1/academics/projects/659-report.pdf.

Hennessy, J., Patterson, D., Computer Architecture: A Quantitative Approach, 3rd ed. San Francisco, CA: Morgan Kaufmann, 2003. pp 41.

Hifn, "Hifn HIPP Security Processor 7855," 2005. [Online] Available: http://www.hifn.com/docs/7855_8_05.pdf. Accessed August 30, 2005.

Hodjat, A., Verbauwhede, I., "High-Throughput Programmable Cryptocoprocessor," IEEE Micro, 24 (3), 2004, pp. 34-45.

IBM, "UltraCypher CRYPTOGRAPHIC ENGINE," 1998. [Online] Available: ftp://www6.software.ibm.com/software/cryptocards/ultracypher.pdf. Accessed August 30, 2005.

InfoAssure, Inc., "InfoAssure – About / Mission," [Online] Available: http://www.infoassure.net/mission.html. Accessed July 23, 2005.

InfoAssure, Inc., "InfoAssure – Products / Mission," [Online] Available: http://www.infoassure.net/need2know.html. Accessed July 23, 2005.

InfoAssure, Inc., "Need2Know Cryptography," Proprietary. August 26, 2004.

McLoone, M., McCanny, J., "A Single-Chip IPSec Cryptographic Processor," IEEE Workshop on Signal Processing Systems (SiPS) Design and Implementation,

California, 2002. [Online] Accessible: http://ieeexplore.ieee.org/iel5/8127/22481/01049698.pdf. Accessed August 30, 2005.

Mingyu, F., Jinahua, W., Guangwei, W., "A Design of Hardware Cryptographic Co-Processor," Proceedings of the 2003 IEEE Workshop on Information Assurance, West Point, NY, 2003.

Mosanya, E., Teuscher, C., Restrepo, H., Galley, P., Sanchez, E., "CryptoBooster: A Reconfigurable and Modular Cryptographic Coprocessor," CHES 1999. Lecture Notes in Computer Science. [Online] Available: http://www.teuscher-research.ch/download/christof/papers/mosanya99_ches99.pdf. Accessed August 30, 2005.

National Institute of Standards and Technology, "AES Key Wrap Specification," 2001. [Online] Available http://csrc.nist.gov/CryptoToolkit/kms/AES_key_wrap.pdf. Accessed April 20, 2005.

National Institute of Standards and Technology, "FIPS 197: Advanced Encryption Standard (AES)," 2001. [Online] Available http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf. Accessed April 20, 2005.

National Institute of Standards and Technology, "FIPS 180-2: Secure Hash Standard," 2002. [Online] Available http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf. Accessed April 20, 2005.

Paar, C., Chetwynd, B., Connor, T., Deng, S. Y., Marchant, S., "An Algorithm Agile Cryptographic Co-Processor Based on FPGAs," SPIE Symposium on Voice, Video, and Data Communications, Boston, MA, 1999. [Online] Available: http://www.crypto.ruhr-uni-bochum.de/imperia/md/content/texte/brendonpaarspie99.pdf. Accessed August 30, 2005.

SafeNet, Inc., "SafeXcel-1840 High-Performance Security Co-Processor," 2005. [Online] Available: http://www.safenet-inc.com/Library/3/SafeXcel-1840_ProductBrief.pdf. Accessed August 30, 2005.

Ravi, S. et al., "System Design Methodologies for a Wireless Security Processing Platform," Proc. 39th Design Automation Conf. (DAC 02), ACM Press, 2002, pp.777-782. [Online] Available: http://www.princeton.edu/~npotlapa/files/papers/dac02.pdf. Accessed August 30, 2005.

Stallings, W., Cryptography and Network Security: Principles and Practices, 3rd ed. Upper Saddle River, NJ: Prentice-Hall, 2003. pp. 1-17, 24.

Stallings, W., Cryptography and Network Security: Principles and Practices, 3rd ed. Upper Saddle River, NJ: Prentice-Hall, 2003. pp. 258-268.

Wu, L., Weaver, C., Austin, T., "CryptoManiac: A Fast Flexible Architecture for Secure Communication," Proc. 28th Int'l Symp. Computer Architecture (ISCA-01), IEEE CS Press, 2001, pp. 110-119.

Xilinx, "Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet," v4.4, September, 2005, pp. 54. Available: http://direct.xilinx.com/bvdocs/publications/ds083.pdf. Accessed October 4, 2005.

Vita

Scott Edward Fields was born in Oak Ridge, TN, on March 6, 1981. He attended Karns High School in Knoxville, TN and graduated as valedictorian in 1999. He then studied Computer Engineering at the University of Tennessee in Knoxville. During his undergraduate study, he completed a co-op with Adtran, Inc. in Huntsville, AL, where he first used programmable logic devices. He earned his Bachelor of Science degree Summa cum Laude in 2004 and is currently pursuing his Master of Science degree in Computer Engineering.