A SECURE ADAPTIVE NETWORK PROCESSOR

Scott J. Harper

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

Doctor of Philosophy
In
Electrical Engineering

April 30th, 2003
Bradley Department of Electrical and Computer Engineering
Blacksburg, Virginia

Keywords: Secure Hardware, Network Processing, Reconfiguration

A SECURE ADAPTIVE NETWORK PROCESSOR

Scott J. Harper

(ABSTRACT)

Network processors are becoming a predominant feature in the field of network hardware. As new network protocols emerge and data speeds increase, contemporary general-purpose network processors are entering their second generation and academic research is being actively conducted into new techniques for the design and implementation of these systems. At the same time, systems ranging from secured military communications equipment to consumer devices are being updated to provide network connectivity. Many of these devices require, or would benefit from, the inclusion of device security in addition to data security. Whether it is a top-secret encryption scheme that must be concealed or a personal device that needs protection against unauthorized use, security of the device itself is becoming an important factor in system design. Unfortunately, current network processor solutions were not developed with device security in mind. A secure adaptive network processor can provide the means to fill this gap while continuing to provide full support for emerging communication protocols. This dissertation describes the concept and structure of one such device. Analysis of the hardware security provided by the proposed device is provided to highlight strengths and weaknesses, while a prototype system is developed to allow it to be embedded into practical applications for investigation. Two such applications are developed, using the device to provide support for both a secure network edge device and a user-adaptable network gateway. Results of these experiments indicate that the proposed device is useful both as a hardware security measure and as a basis for user adaptation of information-handling systems.

ACKNOWLEDGEMENTS

Many thanks go to the Virginia Tech Bradley Department of Electrical and Computer Engineering and its faculty, staff, and students that have made my time spent here both rewarding and enjoyable. I would like to thank all of the members of the Configurable Computing Lab for always keeping things interesting, particularly those with whom I have worked in the past and those who have contributed to the effort of the Secure Communications Group. May all of our work find a host of useful applications.

This work was funded in part by the United States Office of Naval Research (ONR) under grant #N00014-00-1-0549 *– the NAVCIITI effort. I would like to thank the ONR for their continued support of cutting-edge research.*

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

"Learn as much by writing as by reading."
- Lord Acton

THE INTRODUCTION

Network processors are becoming a predominant feature in the field of network hardware. As new network protocols emerge and data speeds increase, contemporary general-purpose network processors are entering their second generation and academic research is being actively conducted into new techniques for the design and implementation of these systems. At the same time, systems ranging from secured military communications equipment to consumer devices such as cellular telephones and PDAs are being updated to provide network connectivity. Many of these devices require, or would benefit from, the inclusion of device security in addition to data security. Whether it is a top-secret encryption scheme that must be concealed or a personal device that needs protection against unauthorized use, security of the device itself is becoming an important factor in system design. Unfortunately, current network processor solutions were not developed with device security in mind. A secure adaptive network processor can provide the means to fill this gap while continuing to provide full support for emerging communication protocols.

1.1 Scope

This dissertation outlines the development of a network processing system that is capable of securely reconfiguring itself while running – a secure adaptive network processor. This effort is driven by the need for flexibility and performance in network data-processing systems as well as the need for security in the hardware supporting these systems. The solution presented here provides a high degree of both flexibility and security without sacrificing system performance. To demonstrate these features, a secure processor prototype is constructed and embedded in a network system where it is subjected to real network traffic for experimental evaluation.

1.2 Problem Overview

Network system development and consumer appetite for information-handling devices are both evolving at a rapid rates. As this evolution occurs, manufacturers are continuously producing new custom hardware to handle the latest protocols and process content for the end user. The end users are keeping abreast of the wave by constantly purchasing new hardware to handle the hottest fashion and provide the most recent features. Both are interesting in protecting their hardware investment and would like to see that investment provide a more lasting service.

Portable devices, including cell phones, Personal Digital Assistants (PDAs), and laptop computers are playing an increasing role in the lives of consumers. As these devices become more integrated with the daily routines of the end-users, they are more likely to contain more personal information, including telephone numbers, birthdates, contact lists, and perhaps even credit card, bank account, or other billing information. Loss of a device containing this type of information is not only inconvenient; it can easily lead to identity theft [1]. As the theft of devices like cellular phones becomes more common [2], end-users would like to see products that not only contain the latest features, but also provide some assurance of security. For example, a cellular telephone that allows a user to change providers and technologies without buying new hardware would certainly find a place in the pockets of consumers. These same users would also derive comfort from the idea that this leading-edge adaptable communication hardware is keyed to them personally and will not function should it be lost or stolen.

Manufacturers would also welcome network hardware that conceals their intellectual property while providing a simple path for patching and upgrading their technology. With a stable hardware platform, they could focus on adding the features that set them apart from the competition.

As this evolution in communication systems takes place, network hardware options are evolving as well. Older network systems relied upon application-specific integrated circuits (ASICs) to provide processing of network traffic. These ASICs provide the high throughput required, but lack the flexibility needed to support changing network protocols. From a security standpoint, ASICs also fall short, as they can be disassembled at any time to discover their contents. This type of an attack on a system not only causes a loss of system integrity, but also results in a loss of the intellectual property contained within the device.

Newer network systems require programmable hardware resources and some have turned to general-purpose processors. Support for network protocols and data rates, however, require significant processor overhead and, as outlined in Section 2.1, there is little indication that contemporary general-purpose processors operate fast enough to support increasing throughput requirements. In addition, processor-based systems are susceptible to security attacks based upon known processor functions and operational monitoring. Exploitation of known processor weaknesses might allow attackers to interfere with system operation. Attacks made against the power connections of system based upon a general-purpose processor can provide information regarding processor data content, including secret keys [3,4,5]. Monitoring of processor connections allows interested parties to easily replicate the system, taking with them the intellectual property of that system.

In an effort to provide better support for modern network systems, researchers have begun developing specialized network processors. The resulting processors range from pattern processors and Ethernet classifiers [32,33,34] to systems consisting of general-purpose RISC (Reduced Instruction Set Computer) cores [21,24,26,29] or reconfigurable hardware with additional special-purpose coprocessors designed to assist in common network packet processing tasks [31,35,36,38,41] (see Section 2.3). In general, these network processors are intended to provide more flexibility than custom Application-Specific Circuits (ASICs) coupled with more efficient handling of network traffic than general-purpose processors can provide. Network processors also serve to prolong networking hardware lifetimes beyond ASIC solutions since they can be programmed to support alternate protocols or modifications to existing protocols.

Unfortunately, most of the currently available network processors have not been developed with security in mind. Current devices are designed primarily for efficient packet routing and high throughput, and remain somewhat limited in their flexibility and lack native data processing (encryption, compression, etc.) capabilities. Those devices that do provide some security or data processing support provide only limited support for data encryption. Additional processing support must be added in the form of a coprocessor, while no attempt to secure the device itself against unwanted use or reverse engineering is made. In addition, these processors often are based upon an existing microprocessor core that does not include a direct hardware implementation of instructions that are fundamental to the task at hand. This dependence on a fixed instruction set provides a readily accessible programming interface, but limits both the potential performance gains and the flexibility of the network processor. It can also present a security concern as the processor language is predefined and the instruction stream can be monitored and interpreted.

1.3 Solution Synopsis

A reconfigurable processor can provide both the flexibility and performance required in current cutting-edge and future network applications while adding a significant level of device security. It can provide custom data-flow processing of network traffic, allowing for on-the-fly encryption and other in-packet data processing tasks, and can adapt its structure to support new fundamental processing tasks required by a particular network. Furthermore, the device can adapt to support individual user needs and be reconfigured after the user is gone to remove all traces of that support from prying eyes. This degree of flexibility serves to increase both the performance and security of the device by allowing for direct hardware support of network protocols and concealment of that support.

By allowing processor reconfiguration to take place at run-time, a network node gains the ability to adapt fluidly to new data handling requirements. Without completely halting information flow within a node, the network processor can update existing data handling routines or add new ones. Run-time reconfiguration of the processor hardware also allows hardware modifications to be specified within the network traffic, eliminating the need to manually locate and modify individual nodes when an update is required.

The ability of a processor to reconfigure itself at run-time not only enhances data-handling capabilities, but also allows for an advanced degree of hardware security. This type of device allows for construction of a system that contains only a generic user discovery application when not in use. The system can adapt its data handling functions to support a particular individual while they are using it, then return to a generic state when the are done. This type of system resists reverse engineering attempts, as it only contains information pertaining to its operation while a valid user is present.

The investigation and development of one such adaptive network processor prototype is presented here. With a focus on the added benefit of hardware security, a prototype system based upon Field Programmable Gate Arrays (FPGAs) is developed and tested in a network environment.

1.4 Contributions and Claims

The objectives of this effort are (*a*) to demonstrate the viability of a secured run-time reconfigurable processor for streaming-data communication applications, and (*b*) to develop structured methods for assembling and reconfiguring this type of processor. The processor prototype is developed on a modified commercial FPGA platform and demonstrated in a wired/wireless network system. The primary challenge faced in this effort is the development of a processor structure that supports secure run-time reconfiguration and, more fundamentally, allows its hardware configuration to be modified while running. To support this effort, a technique for obtaining and utilizing hardware reprogramming information was developed and a method for providing reconfiguration information to the system was created. The primary benefits attained through this, which advance the state-of-the-art, are a manageable reconfigurable network processing system and protection of the intellectual property contained within that system. The concept of system security presented here is not limited to network processing, however. It may be directly applied to any processing system that requires enhanced protection from reverse engineering.

The strengths and weaknesses of the working prototype are analyzed and experimental results are examined to determine the effectiveness of the implementation. The resulting contributions of this work are fourfold:

1. A secure system structure capable of supporting reconfigurable network processing is developed.

2. A key management technique for securing network processor assembly data and its transfer to the system is provided.

3. A simple user-customization and upgrade path for the underlying hardware of a network processing system is defined.

4. A prototype is produced to demonstrate and evaluate the secure network processing concepts introduced above.

As Chapter 2 more fully discusses, the ability to modify hardware within a processor at run-time provides a significantly greater degree of hardware security than traditional ASIC or microprocessor-based approaches. In addition, it considerably extends the life of network hardware by allowing it to adapt fully to protocols that have not yet been conceived while providing the potential to handle data more efficiently than systems based upon microprocessors.

1.5 Dissertation Outline

This document describes a solution to the problem presented in Section 1.2. Chapter 2 provides background information as it reviews work that has been done in the fields of network processor design and secure embedded hardware. In Chapter 3, an overview of the structure and operation of the secure run-time reconfigurable network processing solution is presented. Chapter 4 details a physical implementation of the solution in a prototyping platform. Experiments using this prototype as a secure configurable wireless network gateway and as a wireless end-user network interface are discussed in Chapter 5. Finally, Chapter 6 summarizes the work and provides some insight into future directions.

CHAPTER 2

"How can we accept another to keep our secret if we have been unable to keep it ourselves?"
- Francois De La Rochefoucauld

THE BACKGROUND

As network data rates and processing requirements continue to increase, specialized network processors are being developed as an alternative to the general-purpose processors and Application-Specific Integrated Circuits (ASICs) traditionally found in network data handling systems. These network processors are designed to provide systems with both ASIC-like speeds and microprocessor-like flexibility. Although the presently available commercial devices are primarily designed much like augmented RISC machines, current research is investigating the use of reconfigurable logic to provide additional speed and flexibility to end product designs.

As network devices become more prolific, applications requiring enhanced device security will arise. Military use of network systems for encrypted data traffic might depend on algorithms that are not publicly available. It is desirable for the structure of these systems to remain hidden. In the commercial realm, it is easily within the foreseeable future that applications such as cellular telephones customize themselves on a per-user basis. Reconfigurable processing provides an excellent basis for this, and enhanced device security can serve to protect corporate investment in the intellectual property that goes in to the technology. From a consumer viewpoint, it is also desirable to protect these new devices from unauthorized use. Enhanced security within the device itself provides user-specific customization without creating a device that is easily modified for unauthorized use.

This chapter provides an introduction to network processing and embedded device security in Sections 2.1 and 2.2, respectively. Section 2.3 contains an overview of the current state-of-the-art in the area of network processing. Section 2.4 summarizes the findings, and defines the security shortfall in current network processor approaches. A solution to this shortfall is described in depth in Chapter 3, and is the focus of the remainder of this document.

2.1 Network Processing

Traditional network data-handling systems (switches, routers, edge devices, etc.) use either ASICs for performance [6,7] or general-purpose processors for flexibility and cost reduction [8,9]. Unfortunately, these devices do not offer the combination of speed and flexibility sought in current network systems [10,11]. ASICs provide high-speed data processing, but do not contain sufficient flexibility to adapt to new protocols and features [10]. Additionally, ASIC development is a costly and time-consuming process, requiring significant numbers of ASIC designers and millions of dollars in foundry fees just to produce a single device [10,11,12]. This level of investment in a device is a limiting factor to innovation, and can cause even the most well established players to hesitate before updating an established technology or pursuing a new direction. General-purpose processors are quite flexible, but do not offer the performance required for some network system tasks [10]. At the current rate of growth for both network interface and general-purpose processor speeds, this performance gap is continuing to widen [10]. These issues have created a demand for specialized network processors.

To meet this demand, several network processor solutions have been developed. These hardware solutions range from full single-chip network processors to individual special purpose programmable network processing devices, all falling into one of the following general categories: *general-purpose* processors and *focused-function* devices.

A *network processor* is defined here as a re-programmable processing device that is designed specifically for use in a networking environment. It must be able to provide computing services for the continuous data flows that comprise a networking environment.

To accomplish this, its instruction set or architecture is tailored to tasks (such as pattern matching, traffic classification, buffering, and queuing) that are common in a network environment.

2.2 Embedded Device Security

Device security becomes a concern as network-attached devices become more ubiquitous and competition among vendors increases. For some applications, such as military communications, it is important to protect the network hardware itself from attempts to reverse engineer or replicate its operation (or from incompetence). From the manufacturer's standpoint, it is also important to protect the intellectual property (IP) investment contained in a network product. Gathering the core system function into a single device limits the number of exposed interfaces that can be monitored to determine system function. Organizations involved in proprietary network data manipulation can take advantage of this by encapsulating all IP functionality within a network processor. Unfortunately, current network processors (see Section 2.3) are not designed to meaningfully provide this type of security.

Solutions based upon conventional ASIC and processor techniques fall short of providing the full protection for the algorithms they contain. ASICs retain their functionality even when the system is not operating. As a result, an ASIC can be opened and studied to determine its internal structure [13]. A conventional processor may not retain information regarding tasks when not in operation, but it is vulnerable to bus-monitoring attacks while running [14]. In a processor-based system, the IP investment is in the code fed to the processor at run-time. Monitoring device pins allows this code to be copied or modified. Knowledge of the processor internals allows for the copied code to be decompiled into the algorithms used by the system.

A reconfigurable logic device, such as a Field Programmable Gate Array (FPGA), does not need to retain its function while the system is not in operation. When unpowered, the FPGA is a generic reconfigurable device containing no user-specific function. While operating, it can remain a generic piece of hardware running a simple user discovery application until a valid user presents credentials to the system. Once a user is validated by the system, the device can be configured to include functionality tailored to that user. While common FPGA configuration streams are clear text, protection of the information using a standard cryptographic protocol can provide for a measure of security as this data is transferred to the device [15,16]. When a user has finished with the system, the device can be reprogrammed to once again contain an innocuous application, removing all traces of the user function. If power is removed, even this user function is lost, thus protecting against offline ASIC-like reverse engineering. The fact that all functional data is transferred at once, prior to operation, also reduces the susceptibility to bus-snooping attacks while in operation.

In both processor-based and FPGA-based systems, program information is often held in static random-access memory (SRAM). In both cases, a secure application must take care to eradicate itself from the RAM, since it may be possible to detect the residual information in SRAM cells after power has been removed [17]. Since many FPGA technologies hold configuration information in SRAM, this means care must be taken when a user is done with the system. To prevent recovery of data from the SRAM after a valid user has left, the innocuous application programmed back in must occupy the full region of the device in which the user application was embedded, or at least sufficient area to obscure the user operation.

2.3 Current State of the Art for Network Processing

Network processing is an active field both academically and commercially. Commercial network processors are viewed as the next step towards high-speed data traffic processing and are currently enjoying a second generation as the initial multitude of network processor companies merge together into a few key players. At the same time, academic research continues into new methods of providing network services in reconfigurable hardware. This section identifies the contemporary key corporate network processor entities and discusses current academic research directed toward moving the network processor into reconfigurable hardware.

2.3.1 General-purpose Commercial Network Processing Chips

2.3.1.1 INTEL: Level One IXP Family

In late 1999, the integrated circuit manufacturer Intel Corporation acquired Level One Communications Incorporated in an effort to enter the network processing business. Level One provided several networking solutions including the IXP1200 Network Processor [18]. The IXP1200 consisted of an Intel StrongARM RISC processor and six programmable multithreaded micro-engines. The micro-engines operated using an instruction set specifically designed for networking and communications applications requiring efficient data handling. Operating at 162 MHz, the processor was capable of supporting low-speed switching and routing in real time [19]. Intel currently offers this device in the original format or updated to a 232 MHz package with the functionality unchanged. Additional devices in the family now include the IXP1240 (adding Cyclic Redundancy Check, or CRC, functions) [20] and IXP1250 (adding Error Correcting Code, or ECC, and CRC functions) [21]. None of these processors were designed with device security as a priority, and as such, they remain susceptible to standard microprocessor-based attacks such as bus snooping and RAM monitoring.

More recent products in the IXP line include the IXP2850 [22]: a network processor designed to provide some security function acceleration. This device is built upon the foundation of the previous Intel network processors, adding in hardware support for data security including the Data Encryption Standard (DES), Triple DES (3DES), Advanced Encryption Standard (AES), and early Secure Hash Algorithm (SHA-1). In addition, it supports applications requiring a higher data rate, allowing for multi-gigabit-per-second packet forwarding and traffic management. This device enables designs that must process secured data at high rates by coupling the cryptographic support functions with the network processing hardware. It has not, however, addressed hardware security. Hash table data and the instruction stream itself are stored off-chip, allowing for the use of simple bus-monitoring techniques to acquire data, keys, and program functionality.

2.3.1.2 IBM: PowerNP Family

IBM has been developing network processor technology in-house since the initial release of the PowerNP NP4GS3 [23]. This device is designed to handle multi-gigabit Ethernet traffic, using a PowerPC 405 control processor to govern the actions of sixteen RISC protocol processors [24]. Table data and policing rules are stored on-chip in a 384KB RAM. External dynamic and static RAM is used for large control repositories and data storage. The most recent addition to the PowerNP family is the NP2G [25]. An extension of the NP4GS3 technology, this device reduces the number of protocol processors to twelve, while adding an on-chip switching engine and additional MAC (Medium Access Control) support.

The PowerNP family is not designed to provide inherent support for secure applications. While the use of internal instruction storage does reduce bus transfers, initial loading of this memory can be observed on exposed interfaces. In addition, support for data security would need to be provided by an external coprocessor.

2.3.1.3 Motorola: C-Port Digital Communications Processors

In the late 1990s, C-Port Corporation developed a network processor similar to the IXP1200. Called the Digital Communications Processor (the C-5 DCP), it supported cell and packet processing, table lookup, and queue management [26]. Sixteen RISC-based Channel Processors and five coprocessors provided data processing within the DCP. The channel processors consisted of a RISC core to manage characterization, classification, traffic scheduling, and policy enforcement, with additional send and receive serial data processors to handle field parsing, extraction, deletion, insertion, framing, and CRC checking. The five coprocessors included an Executive Processor (XP), a Fabric Processor (FP), a Table Lookup Unit (TLU), a Queue Management Unit (QMU), and a Buffer Management Unit (BMU). The XP managed the overall DCP and coordinated it with external processors, while the FP was used to scale the device.

Motorola has since acquired C-Port Corporation, where this network processing line has continued to grow. In April of 2002, Motorola announced the introduction of a three gigabit-per-second entry in the C-Port family: the C-3e [27]. This device provides 3 Gbps data handling bandwidth with a flexible architecture including eight external PHYs and on-board Ethernet MACs driven by seventeen RISC (MIPS) processors for packet forwarding and thirty-two additional Serial Data Processors for data processing.

Like the IBM PowerNP family, these devices were not designed for secure applications. All instruction and table storage are off-chip, allowing for monitoring attacks, and secure protocol processing would require external coprocessor support.

2.3.1.4 Vitesse: GigaPOWER and TeraPOWER to IQ2200

In the second quarter of 2000, Vitesse Semiconductor Corporation bought both XaQti and SiTera corporations for their network processing lines. SiTera produced a network acceleration chip set for use with external processors, while XaQti's Active Flow Processors were flow-based dedicated processors that performed specific tasks such as packet processing and parsing [28]. Active Flow processors were based on programmable hardware that was modified via firmware upgrades. XaQti was pursuing two product lines: the GigaPOWER family (several Active Flow Processors coupled with an integrated Ethernet Media Access Controller) and the TeraPOWER family (multiple chips that may be combined to form a network processing solution)[28].

Vitesse's current product line includes the IQ2000 and IQ2200 families of network processors. Unlike their more Fixed-function POWER predecessors, the IQ2200 families are stand-alone devices that provide services for network traffic. The IQ2200 consists of four 400MHz 32-bit RISC processors fed by a classification engine [29]. With a wide range of port options and built-in gigabit Ethernet MACs, the IQ2200 is well suited to stand-alone network applications. Like the IBM and Motorola offerings, these devices were not built with security in mind. As a result, they suffer from the same shortcomings seen in the above systems when used as secure network devices.

2.3.1.5 PMC-Sierra: RM9000

PMC-Sierra produces a wide range of networking products ranging from packet processors and classifiers to MIPS processor systems targeted toward communications. Designed to be part of a network processing solution, the RM9000x2 is their current next-generation processor. It contains dual 1 GHz MIPS processor cores, a 160 Gbps multi-port memory fabric, and a 500 MHz transport interface that allows for 16 Gbps data I/O traffic [30]. While it is in no way itself secure, this device represents the leading edge of general-purpose processor technology for network processing today.

2.3.1.6 Chameleon: CS2112 Reconfigurable Communications Processor

Sunnyvale, California's Chameleon Systems is taking a more generally reconfigurable approach to network processing. Their Reconfigurable Communications Processor (RCP) combines a reconfigurable fabric with an ARC processor, PCI controller, and memory controller to provide a fully customizable stream processing solution. The reconfigurable fabric allows for processing contexts to be interchanged in a single clock cycle, in theory reducing power consumption and increasing device utilization [31,32].

This reconfigurable approach should allow for secure protocol cores to be embedded within the device. In addition, the use of configurable logic, rather than general-purpose processors means that there is no external instruction stream to monitor while the device is operating. The system does, however, remain vulnerable to bus monitoring attacks during the initial configuration sequence. An attacker familiar with the configuration data format could intercept the configuration stream and extract the application structure from it, gaining the Intellectual Property (IP) invested in the application. Furthermore, there is no structure in place to validate a user. Malicious parties might modify the user application and reprogram the device for their own ends.

2.3.2 Fixed-function Commercial Network Processing Devices

2.3.2.1 Lucent / Agere: APP1200 Pattern Processor and APP1400 Routing Switch Processor to APP750NP Classification Engine and APP750TM Traffic Manager

Another company producing network-processing hardware in the late 1990's was Agere Corporation. Its products included both a Fast Pattern Processor (the APP1200) and a Routing Switch Processor (the APP1400) [33]. The pattern processor performed wire-speed recognition, classification and re-assembly of packet traffic. The routing switch processor performed wire-speed queuing, traffic management, traffic shaping, packet modification and re-assembly. Agere was acquired by Lucent in 2001 and then spun back off as Agere a short time later. The current line of processors includes higher-speed versions of the APP1200 and APP1400 – the APP750NP Classification Engine and the APP750 Traffic Manager [34].

As network processing chip sets, not one of these devices acting alone provides the processing needed for most applications. Instead, they are typically combined with each other and additional external processors to provide a full network processing system. Secure data processing applications would require additional coprocessors and result in a hardware implementation that would be more difficult to secure.

Programming of the APP*x* devices is done via a proprietary Functional Programming Language and an Agere Scripting Language. This provides a reduction in the number of lines of code required for a typical application, but limits the system to more specific applications.

2.3.2.2 IDT (Solidum): PAX.core

Solidum Systems has also been working in the network processing realm since the late 1990s. Acquired by Integrated Device Technology (IDT) in September 2002, they continue to provide a unique line of classification processors. Their PAX product line is geared specifically toward the task of classification. The original PAX.core 1000, for example, was a gigabit classification core designed for insertion in other VLSI solutions. It was built using a state machine engine and optional functional units (host interface, checksum, configurable counters) implemented in an FPGA bed. The current line is represented by the PAX.port 1200 and 2500 architectures. These programmable classification processors are optimized for content inspection and regular expression matching at gigabit-per-second data rates [35].

All of Solidum's products are programmed using their proprietary PAX hardware description language. By allowing modification of the fundamental system, rather than just the code running on the processor, these products provide more potential flexibility than processor core–based classification engines. In addition, the use of an FPGA base clearly demonstrates that FPGA technology is suitable for high-speed network processing.

2.3.3 Network Processing Research in Academia

2.3.3.1 USC/ISI GRIP Gigabit Rate IPsec Card

In April of 2002, researchers at the University of Southern California's Information Sciences Institute demonstrated a first-generation giga-bit rate packet processing system implemented in several FPGAs [36]. Coupling a gigabit Ethernet MAC with an FPGA prototyping platform, the researchers sought to address the performance bottleneck imposed by the Central Processing Unit (CPU) in single-processor gigabit networking systems. Packet processing functions including basic routing, encryption, and framing are performed in the reconfigurable components. As shown in Figure 2.1, three separate Xilinx *Virtex XCV1000* [37] FPGAs (X0, X1, X2) are used to process data traffic. In the depicted example, X0 performs packet header processing functions including switching and framing, while X1 and X2 perform data transformation operations such as encryption and decryption. This work clearly demonstrates the utility of reconfigurable logic in next-generation network systems.

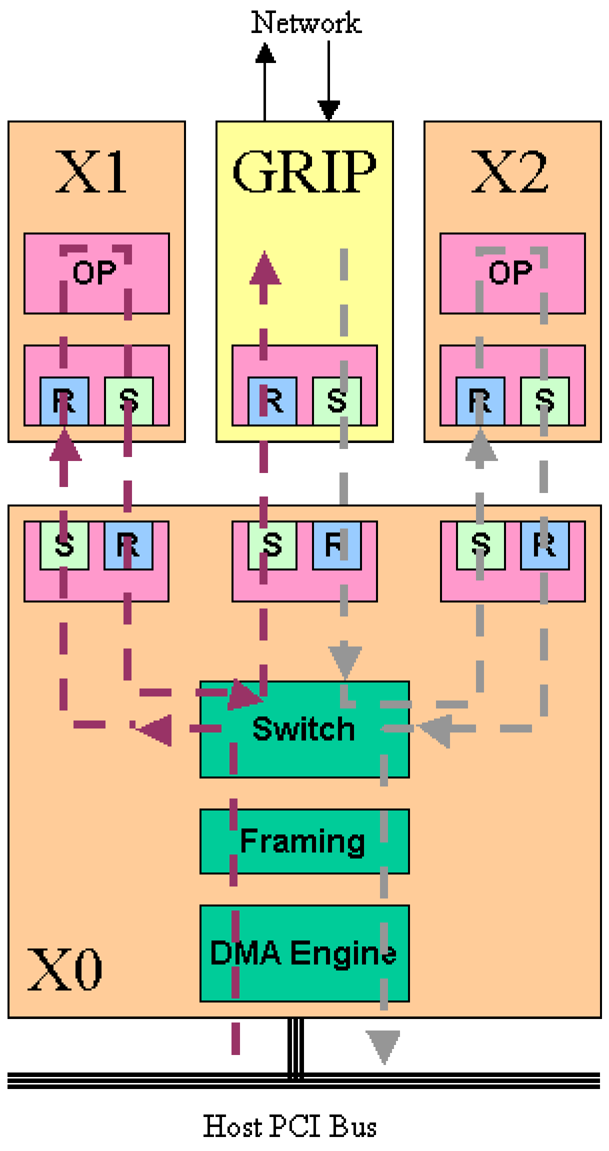


Figure 2.1: *USC/ISI GRIP Gigabit Rate IPsec Card* [36]

2.3.3.2 Washington University Field Programmable Port Extender

A more mature reconfigurable network processing solution is the focus of ongoing research at Washington University in St. Louis, Missouri. The Field Programmable Port Extender (FPX) system uses FPGAs to allow reprogrammable hardware modules to be dynamically installed into network devices [38]. An FPX module contains a static network interface *NID* FPGA (Xilinx *Virtex XCV600E*) and a single reprogrammable user application *RAD* device (Xilinx *Virtex XCV1000E*) as shown in Figure 2.2. These modules are inserted into an existing network stream to provide user-specified traffic processing including routing, buffering, and packet content modification.

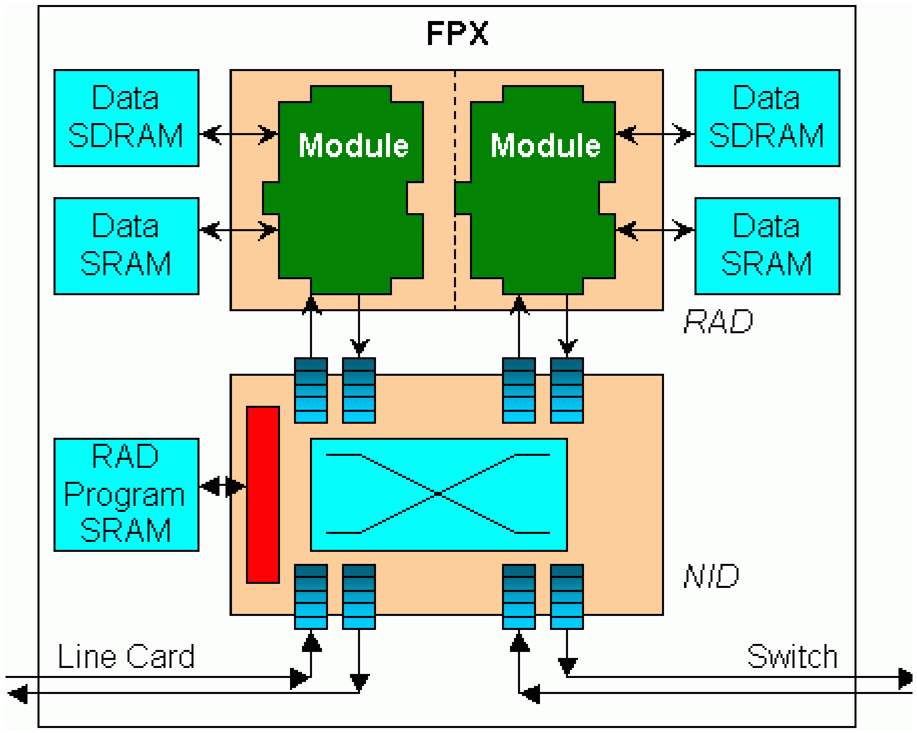


Figure 2.2: *Washington University FPX* [39]

Recent work on this system has included the development of an interface that allows FPX modules to be programmed remotely via a TCP/IP (Transmission Control Protocol over Internet Protocol) network [39]. The NCHARGE tool suite includes both an application programming interface (API) that allows applications to communicate with the FPX and a Web-based user interface to manage control and configuration.

NCHARGE provides no provision for security and the FPX system is not itself secured hardware. The combination does, however, demonstrate the usefulness of an FPGA network processing system and its ability to adapt to network events.

2.3.3.3 UCLA Layer Seven Network Processor Accelerator

UCLA researchers G. Memik, S. O. Memik, and W. H. Smith are also working to advance FPGA-based network processing. They have come up with a reconfigurable OSI Layer Seven accelerator designed to enhance existing network processing applications [40]. Implementing tree lookup and pattern matching algorithms in the reconfigurable logic, they found via simulation that task execution times could be improved by a factor of twenty over the same system implementing the task in software without an accelerator. Within the structure of the simulation experiment, they tested a single-configuration accelerator design composed of a multiplexed set of modules and a design in which new tasks required reconfiguration of the accelerator. Their results indicated that both designs provide a significant performance boost to networking tasks and, moreover, that reconfigurable logic is a viable solution to network processing bottlenecks.

2.3.3.4 University of Edinburgh MIR Mixed-version IP Router

In 2002, Gordon Brebner of the University of Endinburgh [41] described a gigabit IP router that fits on a single Xilinx *Virtex-II Pro XC2VP7* [42] FPGA device. This router takes advantage of the configurability of the FPGA to maintain system performance as traffic migrates from IPv4 to IPv6. Although the work preceded the *Virtex-II Pro* release, and was therefore not implemented in hardware, results did indicate that a prototype would be capable of operation at gigabit Ethernet rates. The results obtained by Brebner clearly pave the way for a single-chip reconfigurable network processor solution.

2.4 Background Summary

Network processors are becoming more predominant in the field of network hardware. As new network protocols emerge and data speeds increase, this trend can only continue. Several approaches to the design of a network processor currently exist. All of these approaches provide some means efficiently handling high-speed network traffic and adapting to new protocols. The current design techniques used to build these network processors do not, however, fully conceal the internal algorithms, nor do they provide support for security of the hardware itself. When the reputation of a network company is based upon its ability to process data more efficiently than the competition, it is important to protect the intellectual property of a design. When that design is critical to the security of the user, it is important that that security not be breached.

As these systems find their way into applications in which the algorithms themselves need to be provided the utmost security (military encryption, etc.), it becomes more important to conceal the device function and control user access. Toward this end, a new scheme is needed to maintain the speed and flexibility of network processors while providing an additional level of device security. An ideal device provides an efficient means of run-time reconfiguration to support leading-edge network protocols, adapts as these protocols change, and implements an efficient method for concealing the internal operation and configuration/programming data presented at its ports. Chapter 3 describes one such system.

CHAPTER 3

"Vision – It reaches beyond the thing that is, into the conception of what can be. Imagination gives you the picture. Vision gives you the impulse to make the picture your own."
- Robert Collier

THE CONCEPT

Several approaches to network processor design were described in Chapter 2. Although each design had something to offer, it was shown that none of them provided sufficient device security to fully protect network systems from intellectual property theft or malicious use. In this chapter, a new approach to network processor design is presented that provides for enhanced device security. This design allows for the flexibility and performance required of the network processor while providing an enhanced level of device security.

Section 3.1 presents an overview of the design, while a general description of the components of this new network processor system structure is contained in Section 3.2. Detailed description of a specific implementation may be found in Chapter 4.

3.1 Processor Overview

A significant amount of current research has been directed toward the design of reconfigurable network systems, as illustrated by the examples in Section 2.3.3. Past work at Virginia Tech has also shown that it is possible to rapidly develop fully functional processor prototypes in reconfigurable hardware [43], and has demonstrated the usefulness of this technology for processing continuously streaming data [44] as well as for handling network traffic [45]. The secure network processor concept presented here expands upon that work to define a network processor structure that provides a level of hardware device security beyond that available using the processors surveyed in Chapter 2. The new structure takes advantage of reconfigurable hardware to allow for both user-based operation and concealment of the intellectual property (IP) contained in a design. Due to its malleable hardware nature, the resultant system benefits from an extended product life cycle, an easy upgrade path, and an inherent user customization capability. It is worth noting that while the full security aspects of this type of processor may not be useful in all network processing systems, the additional benefits to system upgradeability and customization may be useful in most installations that can trade off any added up-front expense of reconfigurable hardware for an upgrade path that does not require hardware modification.

This new level of device security is, in part, achieved by taking advantage of run-time hardware reconfiguration to structurally divide the processor function into two modes – *user discovery* and *user function*. In the *user discovery* mode, the network processor contains a simple application that waits for a valid user to arrive. When the user arrives, this application acts as part of a security system to establish the validity of that user and determine the desired device functionality. Upon user verification, the device reconfigures itself with the protected application and enters *user function* mode. Should the user leave, the device removes all traces of the protected application and once again enters the generic user discovery mode.

The modal division of device operation allows the intellectual property (IP) implemented in specific user functions to be more effectively concealed since the device does not contain the IP when a valid user is not present. While this does not protect against a malicious valid user, it does eliminate the risk of reverse engineering of normal user functionality based upon system structure that is present in ASIC-based devices (see Section 2.2). Furthermore, while a malicious valid user may be able to observe their own system operation, they would not be able to determine the operation of other user systems, nor would they obtain other user credential or biometric information, since the innocuous application embedded between users will remove all trace of the previous user.

Of course the underlying strength of this user-based security is dependent upon the strength of the chosen user validation method. Since the discovery mode application is responsible for user validation and is always available at the device, it could potentially be reverse-engineered by extracting it from local storage and decrypting it without a valid user present. If successful, an attacker will have discovered the local encryption key used for the initially loaded application. As a result, this type of attack could result in the attacker gaining the ability to use the reconfigurable hardware for another purpose. If local keys for initial application encoding are varied between devices, the attacker will not, however, gain the ability to use the hardware of other secure adaptable network processors. Furhtermore, reverse engineering of the discovery application does not provide the credentials or biometric information required to obtain, decode, and embed any user-specific configuration information (Section 3.2). As a result, only a single piece of hardware is lost in the attack, not its function or content. Since the underlying hardware is generally available the attacker could have simply built a similar reconfigurable system with considerably less effort if use of the hardware is their goal. It is, however, possible that an attacker could modify the discovery application, and then entice a valid user to register, thus obtaining that user's configuration information. The encryption used for the user discovery application and the chosen user validation method should, therefore, be resistant to this type of probing in an ideal system.

With an unknown internal structure, bus monitoring or power analysis attacks also become less lucrative. Observation of bus signals while the device is in user function mode may reveal nothing about the device IP since user functionality is wrapped into a single device package that does not require an instruction stream for operation. Similarly, power or timing analysis attacks are more difficult in hardware-based systems than processor-based designs [4]. In processor-based systems, it is difficult to avoid this type of attack by modifying code structure [5]; in a secure adaptable network processor like the one described here, additional circuitry can easily be added at any time to offset (or help mask) timing and power fluctuations. This is somewhat offset by the fact that user configurations must be loaded into the system as users arrive. While bus monitoring of an active system may yield no useful information, monitoring of the configuration operation could potentially reveal the internal device structure. To prevent this sort of attack, configuration must be done in a secure manner.

To achieve the desired level of security, three design maxims must be observed as listed in Table 3.1.

Table 3.1: *Design maxims*

1. **Interfaces that might be subject to observation must be secured.**

2. **Hardware details must be unobservable so that device operation cannot be inferred examination of the device.**

3. **The device must not contain any useful user-specific IP unless a valid user is present.**

In short: ***The system must be resistant to board-level attacks, both online and offline*.**

3.2 Processor Structure

Any description of this network processor design structure must consist of more than just a data processing engine outline. To provide for user identification and functional reconfiguration, external connections must be defined. Furthermore, each of these connections must be secured to prevent observation as per the first design maxim. The system description provided here is divided into four parts: the *user discovery system*, the *user authentication interface*, the *configuration interface*, and the *data processing system*. Section 3.2.1 outlines the operation of the user discovery system, while Section 3.2.2 provides an example of securing the user authentication interface and Section 3.2.3 gives an example of securing the configuration interface. Finally, Section 3.2.4 describes a potential user data processing structure for the network processor.

3.2.1 User Discovery System

In its initial state, this network processor consists of nothing more than a generic reconfigurable hardware device. When power is applied, the system loads a user discovery system through the same encrypted interface that user applications are loaded (discussed later in Section 3.2.3). The state of the powered system prior to user authentication is shown in Figure 3.1, and, as required by the third design maxim above, contains no user-specific function. This system is designed only to recognize valid users and configure the device appropriately. The state diagram of Figure 3.2 outlines the function of the user discovery system.

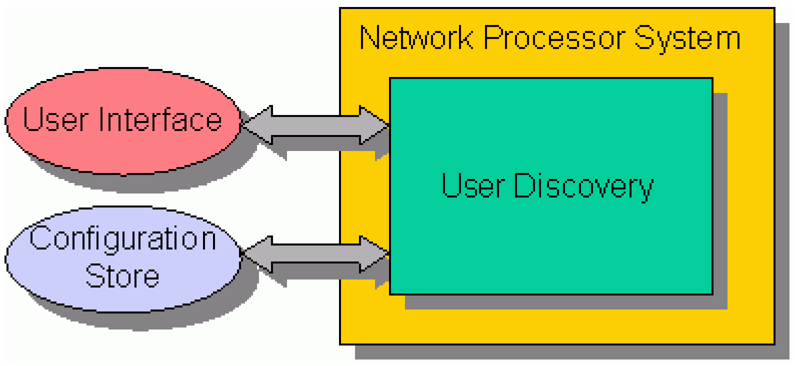


Figure 3.1: *User discovery system*

When a user arrives at the system, they identify themselves through the user interface. Once the system is satisfied that a user is valid (see Section 3.2.2), it determines the required configuration(s) for that user, retrieves them from a *Configuration Store*, decrypts them (see Section 3.2.3), and configures the device for user operation. The resulting user data processing system (see Section 3.2.4) replaces the user discovery system in the network processor, allowing discovery resources to be utilized for the user application.

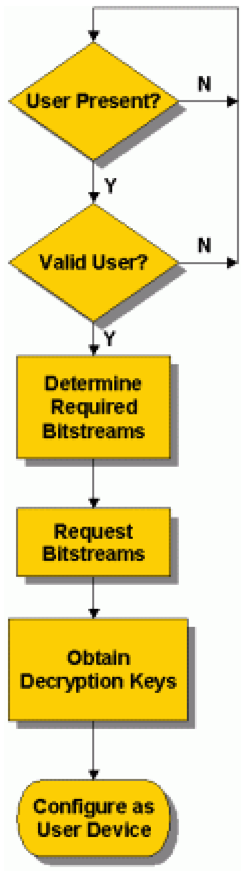


Figure 3.2: *User discovery states*

3.2.2 User Authentication Interface

User authentication depends upon an external interface to the network processor. A user must be able to signal the processor system when they wish to use it and the system must be aware of the fact that they have gone. In the scheme presented here, a biometric signature and a token device identify the user to the system. The token used contains a description of the system desired by the user and sufficient processing power to perform some fundamental security tasks as described in the following sections.

The user authentication interface is exposed to the world and, as per the first maxim, must be protected. The desired level of protection is accomplished by implementing a two-level interface security scheme. At the first level, the network processor and token verify each other and create a secured data path between them. At the second level, the token identifies the user and utilizes the secure data path to provide keys for configuration stream decryption. The token produces these keys using information supplied by the network processor and a set of user credentials that remain in the token. Alternative security schemes may be used for this link, but may result in a decreased security level.

The initial step in establishing user functionality is the creation of a secure link between the user token and the network processing system. When the network processor detects a token insertion, it begins this process by generating a random number, encrypting it with a public key, and sending the encrypted number to the token. Meanwhile, the token verifies the identity of the current user over another secured link (not dealt with here, see [46] for a device that provides a biometrically secured token). When the token receives both the random number and a valid user signal, it decrypts the number received from the network processor and uses it as part of the encryption for a return packet. The returned data identifies the device configuration(s) desired by the user.

This scheme depends upon a public key / private key pair that is embedded in the hardware. In essence, this and the particular encryption scheme used become shared secrets of the platform and token. Guessing the shared secrets by monitoring the signals is difficult in this case, however, as the encryption scheme is not known (and could potentially vary over time or users) and the data transferred from the network processor to the token consists of a random number generated and encrypted internally by the network processor at the start of each user session. Data returned to the network processor from the token is similarly difficult to interpret as it is secured by a scheme that uses the (decrypted) random number initially sent by the network processor. Finally, the secure link should be designed such that it requires an occasional refresh cycle in which the authentication process is repeated to verify that a valid user is still present and to change the key used for data flowing on the link. For details of a particular implementation of this type of scheme, see [47]. For an analysis of some of the weaknesses of this scheme, see Section 3.2.3.3.

3.2.3 Configuration Interface

Upon establishing a secure link to the token, the network processor uses the returned configuration identification information to obtain configuration data from the *Configuration Store*. The Configuration Store depicted in Figure 3.1 will very likely be external to the network processor device. As such, care must be taken to protect both data held in the Configuration Store and the flow of that data from it to the network processor. All configuration data kept in the Configuration Store is therefore pre-encrypted by the developer and then further encrypted using a one-time key and padded with a security header prior to sending it to the network processor. Locally stored user configuration credential information is used by the Configuration Store to generate this one-time key as explained in Section 3.2.3.1. The one-time key generation and padding techniques employed allow for key reconstruction and data decryption by a receiving system that contains the same credential data and is aware of the key reconstruction technique. While it is still possible that the configuration data may be intercepted and decoded, the use of a one-time key guarantees that the effort required for the individual packet decoding process will not provide access to other encrypted configuration data provided by the Configuration Store. Furthermore, the fact that the packaged configuration data is itself encrypted means that the IP contained in the configuration data of the decoded packet still requires additional decoding before it reveals its information. Upon receipt by a valid user, both the packet and the embedded configuration information are decrypted. The configuration information is then used to reconfigure the network processor device. Some potential weaknesses of this scheme are outlined in Section 3.2.3.3. For an example of a commercial scheme similar to the packetization technique used here, see the Constructive Key Management^{TM} (CKM^{TM}) technique developed by TecSec, Incorporated for secure data transfer applications [48].

3.2.3.1 Key Management

The packet returned by the Configuration Store is encrypted with a one-time use key and accompanied by a *security header*. The security header (shown in Figure 3.3) contains an asymmetrically encrypted random number along with symmetrically encrypted tags

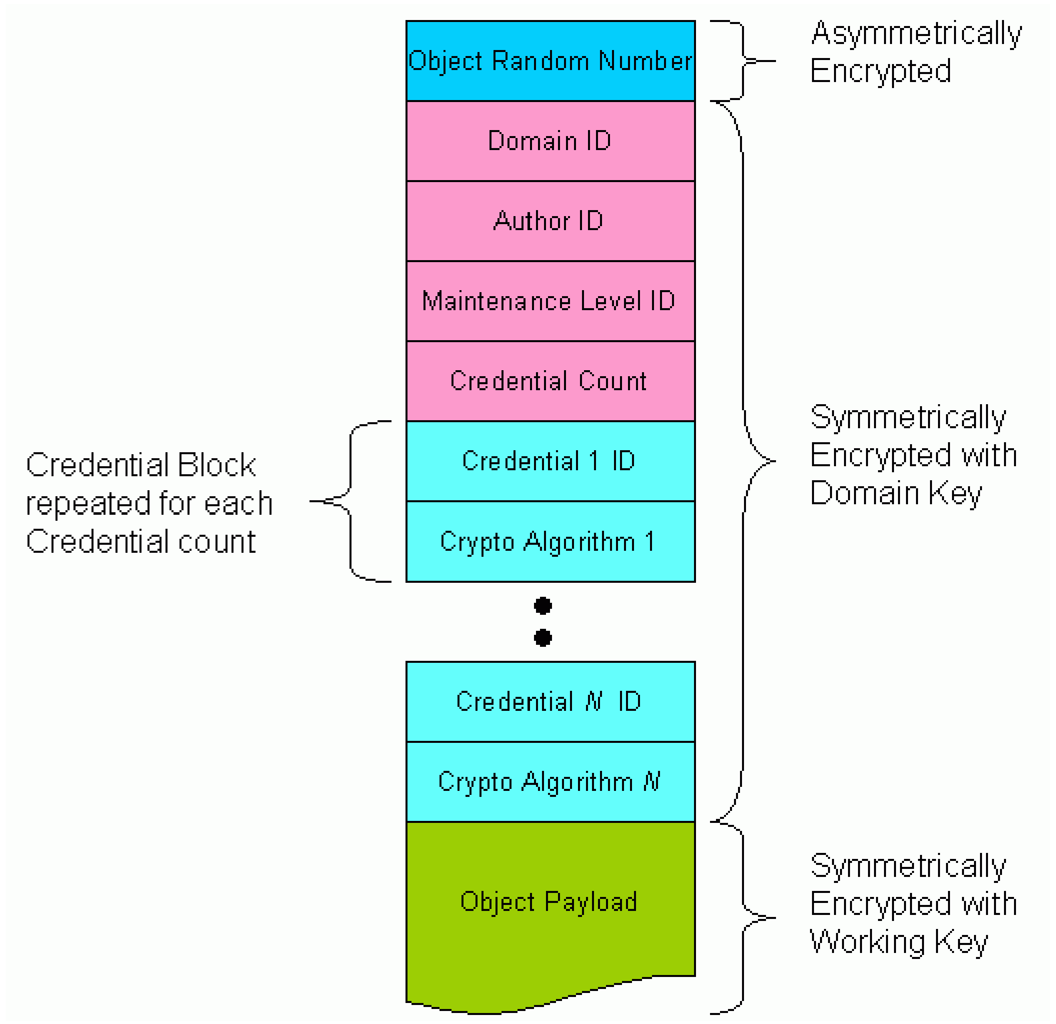


Figure 3.3: *Configuration data security header*

identifying the user domain, author, and number of credentials used. A credential identifier and an identification of the algorithm used to encrypt the random number are included for each credential used in constructing the data set. This header is passed to the token, which processes it to produce a working key for the data. An example of the process by which this happens is summarized in Figure 3.4, where RSA and Blowfish encryption algorithms are combined with a secure hash algorithm to extract the encrypted configuration data from a received configuration object. Once the data has been extracted, it is decrypted using a domain configuration key and the appropriate algorithm to produce a clear-text configuration data stream.

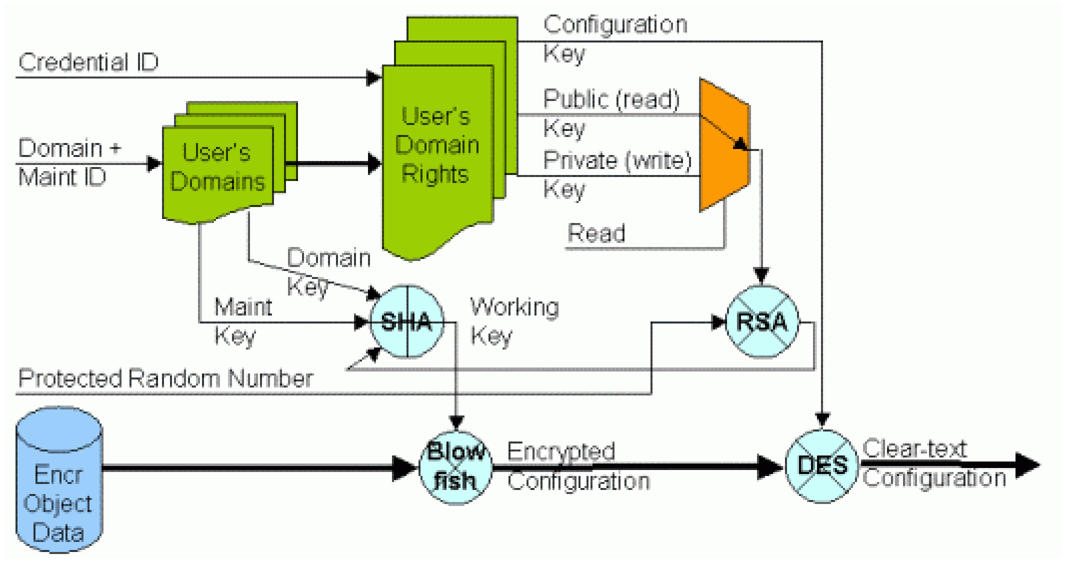


Figure 3.4: *Network processor configuration data reception*

The first element of the data transfer is an *encrypted random number*. This number allows the transfer to have a unique working key that is disposed of after the transfer is complete. The random number is encrypted using the credentials identified later in the header. Decryption of the number allows the recipient to reconstruct a working key required to interpret the data portion of the transfer.

A *domain identifier*, an *author identifier*, a *maintenance level*, and a *credential count* are encrypted with the *domain key* and follow the random number. The maintenance level identifier is a two-part (forward and backward) index into an array of *Maintenance Level Domain Keys*. These keys are shared secrets among member of a domain that can be accessed by a user within the range specified by the forward and backward indexes. The recipient attempts to decrypt this section of the header with each of their maintenance level domain keys in turn. If they have the correct privileges, one of the keys will result in a domain identifier match, which means the correct key has been found.

Given a correct domain key, the credential blocks (consisting of a *credential identifier* and *algorithm identifier*) are decrypted. The shared secret credential key and identified algorithm for each credential are applied in the order given to the encrypted random number to produce the working random number.

A *credential* provides permission to read or write a class of data within a domain, and credentials of this sort allow stratification of information within that domain. Each credential has an identifier that indexes a public/private key pair. This key pair is a shared secret among class members within the domain. The public key provides write permission for a class of data while the private key provides read permission. It is possible to grant a user read permission to a class without write permission. Similarly, a user may be privileged to write data to a class, but may not be allowed to read from that class.

The token contains a *combiner* that uses the working random number along with the domain key to reproduce the one-time key for the transferred data payload. This working key is transferred via the secure link back to the network processor, which uses it to decrypt the data contained in the payload and produce the original encrypted configuration.

Finally, the token passes the appropriate domain configuration key to the network processor and the encrypted configuration is decrypted to produce a clear-text bitstream that is used to configure the network processor for the current user function.

While the proposed key management technique does provide a significant level of information security, it is still susceptible to some limiting factors. Implementation of this technique requires significant processing resources on the token device. The key generation process may potentially require several cryptographic schemes since each credential can be associated with a separate algorithm. This limitation will, however, become less of a factor as technology allows for more processing on token devices. The complexity of the algorithm also limits the speed of the configuration process. In systems where configuration speed is critical, a lighter-weight security system may need to be employed.

3.2.3.2 Token Contents

A domain-oriented approach to key management is employed to promote management of user groups and stratification of users within groups (Section 3.2.3.1). To support this approach, the token must contain the elements listed in Figure 3.5. These elements are placed into a tamper-resistant token device to secure them from inspection. The token must also contain sufficient processing capabilities to perform the tasks outlined in the previous section. One such device token is the Dallas Semiconductor Java-Powered iButton [49]. For an example showing the programming of this device, see [47].

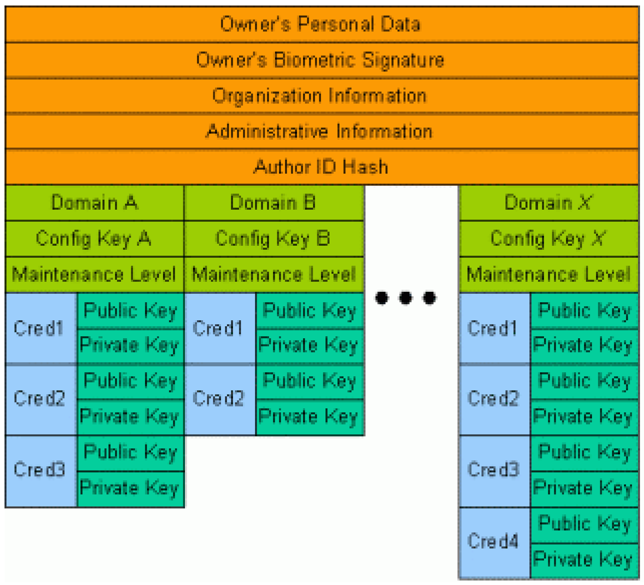


Figure 3.5: *Token data content*

3.2.3.3 Configuration Interface Security Analysis

As indicated previously, this secure network processor is most vulnerable during user configuration. While the proposed authentication, key management, and configuration schemes do provide a significant amount of security, they still remain susceptible to certain types of security attacks.

The weaknesses of this network processor lie primarily in the interfaces. The interface to the Configuration Store is considered to be open to snooping. Data read from it may, in general, be accessed by any interested party and stored for later analysis. It is argued above that this may not pose a significant threat in terms of IP loss. However, should the interface be a general network system, there is the threat of a third party exercising a Denial-of-Service (DoS) attack on a device. Flooding the network processor with data may cause its reconfiguration process to slow dramatically. The network processor does reject any packets that it did not specifically request (or cannot verify), however, minimizing the effect of the attack. A more serious DoS attack would be to flood the Configuration Store with data. Since the Configuration Store is listening for requests from any network processor implementing this scheme, it may take more resources to determine that a request is not from a valid source. Although the Configuration Store will not reply unless a request is validated, operation of the request-receiving port could be hampered. This will affect the configuration times of all legitimate network processors using that storage device. A scheme implementing several distributed configuration storage systems may help alleviate both this type of attack and general loading of a single device.

Perhaps a more serious attack on the Configuration Store interface would be to supplant the system that supplies configuration information with the goal of providing false configuration information to the network processor. The supply of invalid information is not in itself sufficient to configure the device, as the information must be supplied when the device is requesting configuration. The supplied information must also correspond to the requested data in terms of formatting, configuration encryption, and credential-based encoding. Clearly, the success of this type of attack would require intimate knowledge of the key management technique as well as the credentials and domain configuration keys that are applied to various configurations. Using an incorrect technique or credential set would simply result in configuration data that the network processor was unable to decode. As a result, it would fail to configure. Analysis of the Configuration Store itself would not provide sufficient information to produce this information. While full compromise of the Configuration Store would be a serious problem, the configuration data that it contains is encrypted with a set of keys of which it is not aware.

If an attacker where to possess a full set of domain configuration keys and take over a Configuration Store system, configuration information would be compromised. In addition, the attacker could correctly format malicious configuration data and feed it to network processors as data is requested. In similar systems, an authentication server is used to reduce the likelihood of an attack in which a malicious system mimics the response of a valid server. In this case, however, the information supplied by an authentication server would be redundant, as the data exchange format provides its own authentication and would require that the malicious system be a valid configuration server.

The interface used for user authentication is also exposed. The scheme used to provide security at this interface has been designed to eliminate the risk of man-in-the-middle attacks and reduce the effectiveness of brute force attacks. The interface itself is susceptible to flooding, but it is not part of an external network so the risk of this is less than that of someone simply cutting the wires to the interface. For a full analysis of the authentication interface security scheme, see [47].

The strength of the key management security scheme lies in its obfuscation of data keys and protocols and the fact that the secret material required to build the keys and request configurations is neither exposed nor contained within a single system. Brute force attacks to guess these keys required knowledge of the packet structure and, after considerable time and effort, reveal only the key for an individual packet. Data contained within that packet then requires additional decoding before it is useful. Attacks based on intimate knowledge of the system will be successful for all configuration data and will allow for the malicious programming of network devices. However, the threat posed by a malicious entity that knows all credentials, domain keys, configuration data tags, biometric information, data structures, and coding techniques is unavoidable.

3.2.4 Data Processing System

In keeping with the systems described in Chapter 2, one operational data processing engine (user function) for this network processor might consist of an input Classification Engine, several Reconfigurable Processing Channels, and a Post-Processing Engine that arbitrates and/or routes the processed traffic (Figure 3.6). Since the design is targeted toward reconfigurable logic, the Processing Channels may be either streaming pipelines capable of accepting input data at the rate supplied by the Classification Engine or processor-based channels with FIFO queues at the input. Similarly, the Post-Processing Engine may require a FIFO interface or be capable of handling data at the rate it is supplied by the Processing

Channels. For maximal efficiency, the Classification Engine, Processing Channels, and Post-Processing Engine should be able to handle data at the Media Input/Output rates (i.e. not drop packets).

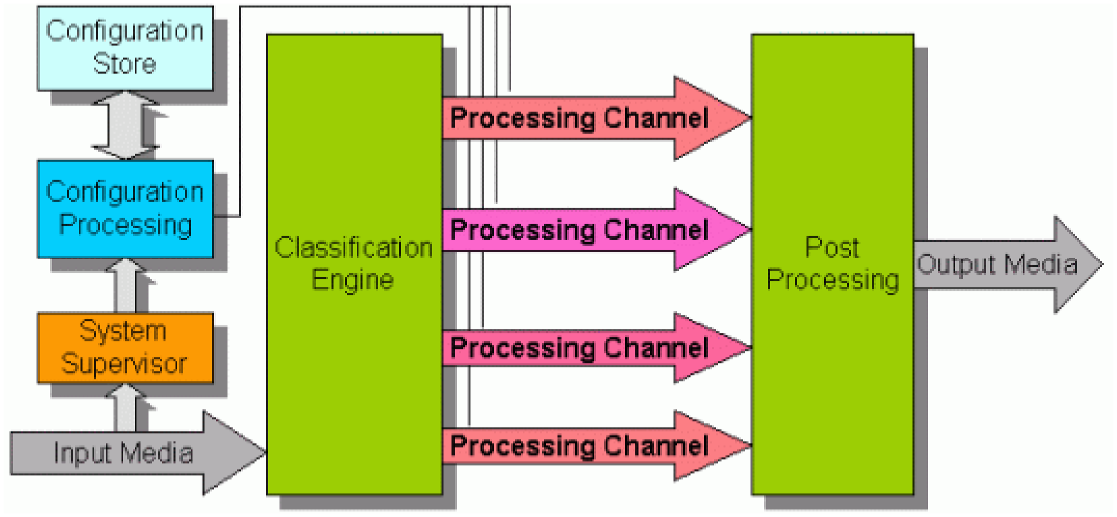


Figure 3.6: *Data processing system (user function)*

Whatever the user function of the network processor, a System Supervisor and a Configuration Processing Unit are needed to oversee device operation. The *Configuration Processing Unit* is responsible for the actual reconfiguration of the operating device, and functions exactly like the configuration state machine found in the user discovery system. If a change is called for, the *System Supervisor* determines which configurations are to be loaded and signals the Configuration Processing Unit appropriately. The Configuration Processing Unit then requests configuration data from the Configuration Store and embeds the represented logic into the configurable elements of the network processor. The System Supervisor is also responsible for verifying user presence while the device is operating. When it detects the user has left, it will initiate a configuration sequence to return the device to a generic user discovery system described in Section 3.2.1.

The *Reconfigurable Processing Channels* provide data processing support for packets moving through the network processor, and may change based upon network traffic. For the purpose of design simplification, run-time reconfiguration is restricted to the Reconfigurable Processing Channels. This simplification allows for pre-established blocks of configurable resources to be dedicated to run-time configuration. Input and output channels are predefined for each block, and replacement channels are required to fit into these parameters. Using this method, partial reconfiguration of the system can modify the operation of a single channel. Additional information regarding device self-reconfiguration is contained in [74].

3.3 Concept Summary

The network processor structure presented in this chapter provides an operational data flow similar to that of the processors summarized in the previous chapter. By carefully implementing the device in reconfigurable logic and adding a small amount of user authentication and configuration support, the device gains a security advantage over those other processors. A bi-modal operation provides security for the intellectual property contained in user functionality, and attention to three basic hardware design maxims provides security for exposed interfaces. In Chapter 4, an example of the use of this type of device is given and an operational version is tested.

CHAPTER 4

"It's scary to think that the infrastructure of the industrialized world is increasingly based on software like this."
– Stephen Wolfe referring to AutoCAD rev. 12 in the 1992 CAD Report p. 11

THE STRUCTURE

The previous chapter described a conception of a secure adaptable network processor. This chapter presents one possible structure that allows that concept to be investigated using contemporary hardware. The structure described in this chapter provides the means for a user discovery application to invoke a function within the network processor hardware. The user discovery application implements the secure user identification, authorization, and configuration aspects described in Chapter 3. The user-specific function then provides data routing and processing capabilities as appropriate for the current user. This structure is then used in Chapter 5 to construct a prototype network processor and to demonstrate its functionality by filling two network positions – a gateway system and an end-user device.

Section 4.1 of this chapter provides an overview of the prototype structure, while Section 4.2 discusses the user discovery application and the means by which the adaptable network processor is securely reconfigured for an end-user application.

4.1 System Overview

The prototype system described in this chapter distributes the network processor function across several configurable devices to allow for run-time reconfiguration and to provide a modular design structure that promotes independent development and validation as well as access points for integration testing (Figure 4.1). Separate configurable elements are used for system configuration control and user processing. Support for both the initial user-specific function and the run-time modification of that function is contained within a single device. A second configurable device contains the interfaces for user hardware and the basic packet handling systems. A third device provides system configuration control and authentication support through during both the user discovery and the user function modes of operation. This division allows for external device (wireless radio-frequency hardware and wired network physical interface) connection establishment during initial user configuration. It also permits run-time updates to the processing channels that do not disturb these external connections.

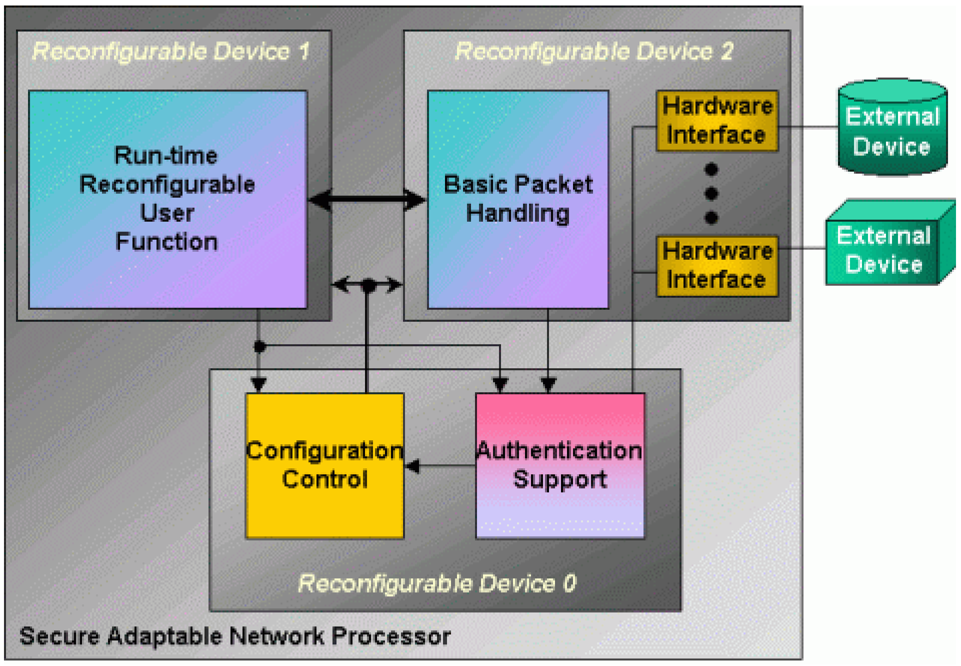


Figure 4.1: *Basic system block diagram*

This multiple-element prototype demonstrates operation of the single-device network processor described in Chapter 3. The use of multiple devices simplifies the initial development and debugging process while allowing for a lower cost prototype. Although the operation has been divided among three components for demonstration, the functionality provided by these three configurable elements could be folded into a single larger reconfigurable device. Additional support for partial run-time reconfiguration must be developed to support single-device packaging, however. Along these lines, configurable logic vendors now provide technologies such as the internal *SelectMAP* [50] interface (*ICAP*) in the Xilinx *Virtex-II* [51] FPGA family. These interfaces allow a device to partially reconfigure itself while it is running. Fong, Harper, and Athanas describe in [74] the current state of ongoing research into this concept of dynamic self-reconfiguration, the results of which will permit the full integration of the prototype described below into a single-device package.

4.2 Configuration Management

As described in Chapter 3, the secure adaptable network processor operates in two distinct modes. When a valid user is present, it provides individualized processing of the user data. When no user is present, a discovery application is run on the hardware. This section describes the system operation as it transitions from user discovery mode to user function mode.

4.2.1 Authorization

In this particular prototype system, a user is identified by a token and a biometric. The user inserts a token into a receptacle attached to the prototype platform and then allows the system to identify them via a biometric interface. The initial authorization process follows the path described in Chapter 3. At the final stage of authorization, information used in assembling the embedded user function is downloaded from the token in the form of a *Bitstream Cache Table*.

This prototype uses a custom-designed *Bitstream Cache Table* to manage the encrypted configuration bitstreams. Each table entry corresponds to an individual bitstream required for current user operation. As shown in Figure 4.2, a table entry contains an identification tag, a decryption key, bitstream memory address ranges, a partial bitstream indicator, and status flags. This format was chosen to allow for modular development of the prototype. As the primary communication point between the authentication system and the configuration engine, it allowed the two systems to be developed independently, then easily merged into a whole system. The table itself is flexible enough to allow for modification of this interface as needed. Without a valid set of table entries, the encrypted bitstream data cannot be retrieved, decrypted, and loaded into processing elements.

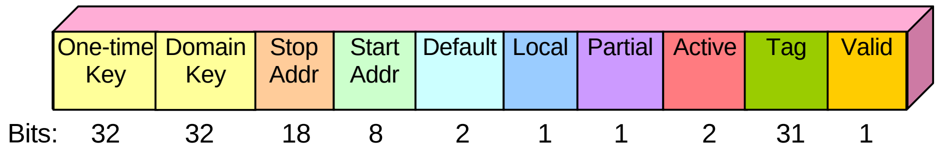


Figure 4.2: *128-bit cache table entry for configuration management*

The *Start Address* and *Stop Address* fields indicate where the encrypted bitstream is stored in local memory after retrieval from a remote server (Section 4.2.3). Entries with the *Default* flag set are to be programmed in as the initial user function. Entries that do not have this flag set represent configuration data that may be swapped in while the network processor is in User Function mode. The *Active* flag is used to mark which configurations are currently being used. After a bitstream is retrieved from the remote server to local memory, it is marked as *Local*. The *Tag* is used to uniquely identify a configuration bitstream and its domain. For *Valid* entries, the previously mentioned fields represent a user application bitstream. Invalid table entries simply represent slots that are not yet filled by the current application. Finally, the *One-time Key* and *Domain Key* fields contain the keys associated with a doubly encrypted configuration entry. The one-time key is determined by reconstruction of the one-time key used by a remote Configuration Server as described in Chapter 3 and discussed further in Section 4.2.3. The domain key is retrieved from the token and used in conjunction with the one-time key to produce a clear-text configuration bitstream. Once these keys are in the table, they are kept inside the secure network processor and used only by the configuration engine (see Figure 4.5).

The Bitstream Cache Table can only be modified by two sources. The first source is the User Discovery application, which transfers the original table data upon user verification. The second source is a *Control Register* in Reconfigurable Device 0 that provides an interface for run-time modification of the system. This register allows the user application to request reconfiguration in addition to providing a debugging interface for the prototype system. Restricting cache table access in this manner allows for internal control of reconfiguration events. Although table access could be restricted further for other applications, the chosen method provides a useful combination of control and accessibility in the prototype system.

Support for user discovery mode is provided by a security management system running inside of the Configuration Engine in *Reconfigurable Device 0* and by an additional *Authentication System* running in *Reconfigurable Device 2*. Additional details of user discovery mode operation are provided in [47].

4.2.2 Stream Management

User bitstreams are retrieved from a network connection based upon the tags placed in the Bitstream Cache Table during the user authentication process. Conversion from User Discovery Mode to User Function Mode is outlined in Figure 4.3. It involves retrieving the appropriate encrypted configuration bitstreams for reconfigurable devices *0* and *1* from an external network and placing them in local storage. The bitstreams are then read by a Configuration Engine, decrypted, and used to place the user application in the network processor.

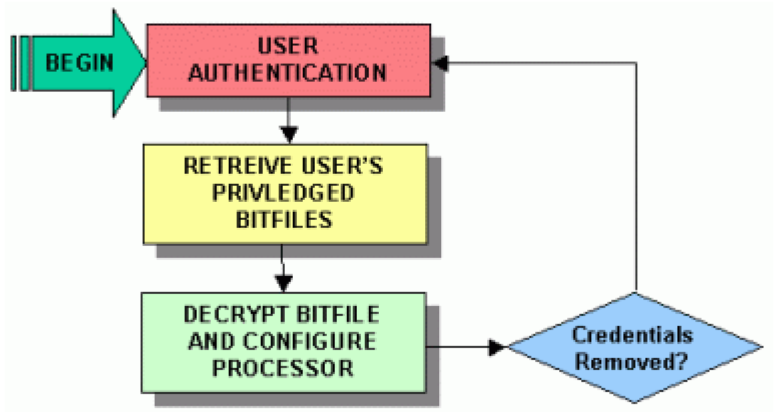

BEGIN
USER AUTHENTICATION
RETREIVE USER'S PRIVLEDGED BITFILES
DECRYPT BITFILE AND CONFIGURE PROCESSOR
Credentials Removed?

Figure 4.3: *Conversion from user discovery to user function mode*

The security management system employed during this operation is shown in Figure 4.4. It consists of a device programmer (Appendix B), a user authentication unit (described in [47]), a network interface, and a configuration manager. The *Device Interface* shown in the picture is a simple register-based connection that directly controls the configuration pins of *Reconfigurable Device 1* and *Reconfigurable Device 2* [52]. Bitstream data is directly written to this interface in unencrypted form to configure the reconfigurable devices that comprise the system. The *Ethernet Interface* may be either a direct (local wired crossover) connection to a display device such as a tablet computer, or a standard connection to a full network system. In the case of a direct connection, the Configuration Store is the connected device.

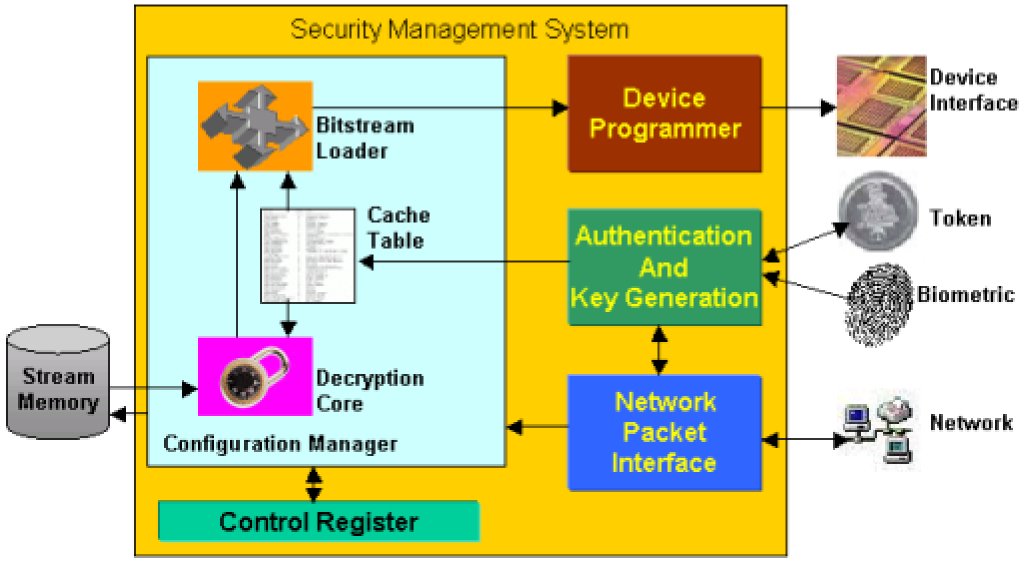


Figure 4.4: *Configuration management*

The *Configuration Manager* is comprised of three major entities: a Decryption Core, a Cache Table (described in Section 4.2.1), and a Bitstream Loader. In this implementation, The *Bitstream Loader* consults the cache table to locate encrypted configuration bitstreams in memory and pass them through the decryption unit to the Device Programmer. The *Decryption Core* decodes the encrypted bitstream using the associated key (provided by the cache table) and returns the result to the Bitstream Loader for forwarding to the Device Programmer.

4.2.3 Configuration Stream Requests

Configuring *Reconfigurable Device 1* and *Reconfigurable Device 2* as a user-specific network processing system changes the platform from a stand-by state to an operational device. As described above, this process is based on the bitstream cache table and handled by the Configuration Manager. When the bitstream cache table is first transferred to the network processor, the Configuration Manager determines if the required encrypted bitstreams are stored locally. For non-local bitstreams, a request is sent over the network interface to a remote Configuration Store to retrieve the needed information as depicted in Figure 4.5.

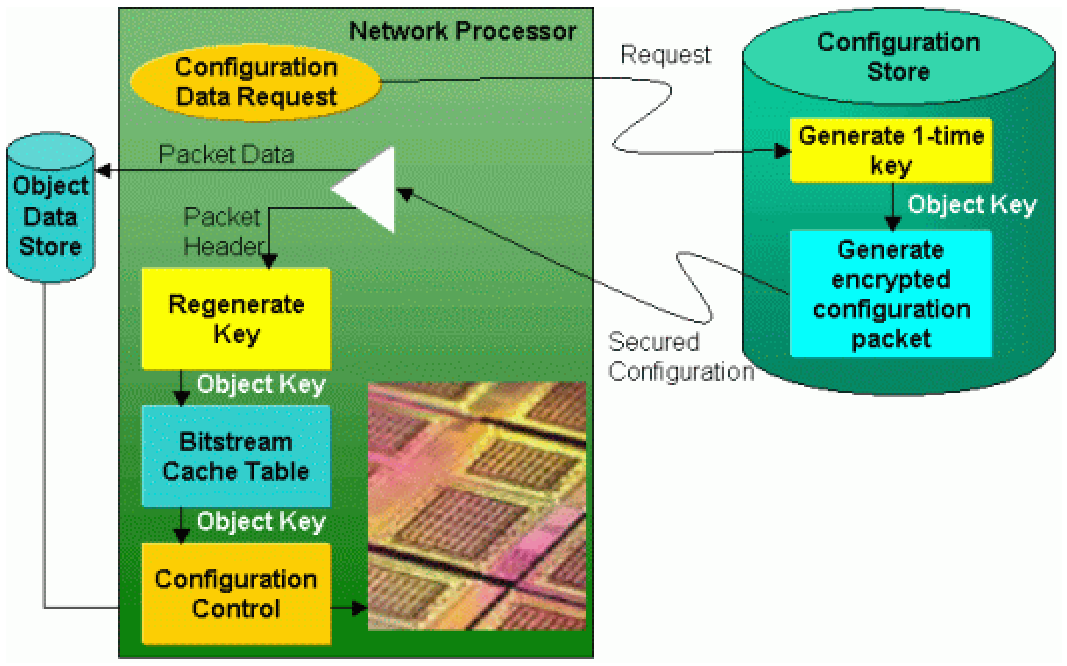


Figure 4.5: *Configuration key handling*

The transmitted request used in this example structure is in clear text and the assumption is made that it does not need to be secured. The request contains only a bitstream identifier. As such, any outside party intercepting it would gain information regarding the bitstream identifiers requested by a particular device. While they would not gain any knowledge of the content of the configuration data, this might provide some advantage in attempting to decode the returned data. In a non-prototype system, this request could be protected using the same scheme that is used here to encrypt returned configuration data. This additional protection of the request would conceal the identifiers, but would also require additional resources. The marginal gain in security was not considered worthwhile for the prototype system, as it does not demonstrate anything outside of the already established protection scheme.

In response to a request, the network Configuration Store returns a packet containing an encrypted bitstream. This bitstream is secured in a packet conforming to the format previously described in Section 3.2.3. Upon receipt of this packet, the network processing system decrypts the packet header and uses a combiner located in *Reconfigurable Device 0* to produce the decryption key for the bitstream. The encrypted bitstream is moved into external storage at the location specified by the key table, while the internal key table is updated with the configuration data key and the bitstream is marked as local.

When all configuration streams are local, the bitstream loader reads those marked as *Default* from local storage, retrieves the one-time key from the Bitstream Cache Table and the domain key from the token, and uses a hardware encryption unit to decrypt the configuration information and forward the results to the appropriate programming interface via the *XVPI_PROG* entity listed in Appendix B.

4.2.4 User-based Reconfiguration

Once the initial configuration is complete, the user-specific system may itself request device reconfiguration. Reconfiguration requests of this type may result from user input or data flowing through the system. The process by which these requests are handled is similar to that of the initial reconfiguration. The bitstream loader is notified by the user application that a configuration update is needed. The update may consist of a bitstream placed in memory by the user application or a tag for a bitstream that must be retrieved from a remote server. In the first case, only the Bitstream Cache Table is updated to reflect the location of the new bitstream and its decryption key. In the later case, the bitstream is retrieved as described above for the initial configuration. In both cases, once the bitstream is local and the table is updated, the bitstream loader decrypts the configuration data and feeds it to the appropriate device configuration interface.

The Bitstream Cache Table keeps track of recently used bitstreams using identification tags to refer to bitstream entries (see Figure 4.2). Bitstream entries may reside anywhere in the table and the table may be extended as needed with the only theoretical limitation being the number of unique identifiers it can contain (2^{31} entries using the current table format). In practice, table access may be restricted to initial configuration only or extended to the user or application for additional device flexibility.

4.3 Structure Summary

The prototype structure described in this chapter incorporates the ideas of Chapter 3 into a system that can be constructed using hardware that is readily available. It provides the flexibility needed to test user applications within the network processor framework and allows for the run-time system modifications that are the basis of the framework. It is not, however, an ideal representation of the network processor. Several steps were taken to produce a realizable system in currently available (and relatively inexpensive) hardware including separation of the functionality across several programmable devices.

An ideal system would incorporate all functionality into a single device like that described in Chapter 3. These changes, however, would come at significant expense when using currently available hardware. Additional resources would also need to be invested to develop reliable self-reconfiguration techniques within the single device. While all of these are reasonable goals in a production system, and certainly worthy of a second prototype, they do not add significantly to the preliminary investigative utility provided by the described prototype platform.

Chapter 5 describes an embodiment of these concepts and structures using currently available hardware. It also presents two generations of the prototype, each of which provide a useful end-user function that is implemented using the secure adaptable network processor.

CHAPTER 5

"You cannot acquire experience by making experiments. You cannot create experience. You must undergo it."
- Albert Camus

THE EXPERIMENTS

In the previous chapter, the basic structure of a secure adaptable network processor prototype was defined. This chapter brings the potential of that structure into focus by investigating two generations of hardware prototypes. The first of these, presented in Section 5.2, is a network edge device that uses the security features of the platform to provide user-specific access to streaming data in a potentially hostile environment. A second prototype is outlined in Section 5.3. It expands upon the initial work to include more of the secure adaptable network processor functionality and to demonstrate a second application that uses the secure network processor system to provide a secured wireless link for a network system. Section 5.1 introduces the chapter by describing the hardware foundation used in both prototypes.

5.1 Foundation Platform

A prototype of this network processor was designed and constructed under the auspices of the Office of Naval Research (ONR) Navy Collaborative Integrated Information Technology Initiative (NAVCIITI) project[1]. Under the provisions of this project, a prototype for a secure wireless network link was to be operationally demonstrated in conjunction with novel wired network and wireless communication systems. The goal of this cooperative effort was to demonstrate enhanced security and flexibility for naval ship-

[1] United States Office of Naval Research, ONR Grant N00014-00-1-0549

to-ship communication systems. For the purposes of this project, each ship was considered to be a single wired network entity with a secured wireless link operating between vessels. A means of providing controlled distribution of classified communication hardware was a desired component of this system. The key management scheme proposed in Chapter 3 provides a mechanism for dynamic control of installations through credential assignment. It allows a site administrator to distribute generic hardware to concerned parties without exposing the underlying technology, to retain control of the operation of that hardware, and to modify the hardware function as needed.

The prototypes described in this chapter demonstrate the concepts of the secure adaptable network processor in a concrete fashion within the practical environment provided by the NAVCIITI project. In addition to a demonstration platform, this development effort provides the opportunity to assess the network processor architecture within the constraints of currently available configurable hardware. Simulation alone could not, in reasonable time, model the fine-grained system control required for this system while performing the functions of the discovery and end-user applications. Furthermore, by working with a physical system during the design phase, rather than simply simulation, it was possible to address unforeseen hardware-related issues while the system was being developed and incorporate lessons learned back in to the development process. As with other rapid prototyping efforts, this experience with hardware early in the development stage allows for a system model that is readily transformable into a practical system.

5.1.1 Reconfigurable Base

The prototype network processor system has been implemented on a SLAAC1-V reconfigurable computing platform [53]. The SLAAC1-V, as shown in Figure 5.1, has three Xilinx *Virtex XCV1000* FPGAs [37] labeled *X0*, *X1*, and *X2*. There are ten independent on-board SRAM memories providing a total capacity of 11.5Mbytes, and a significant amount of inter-FPGA connectivity. In the default SLAAC1-V system, the Configuration Control FPGA (a Xilinx *Virtex XCV300* [37] device) governs device configuration.

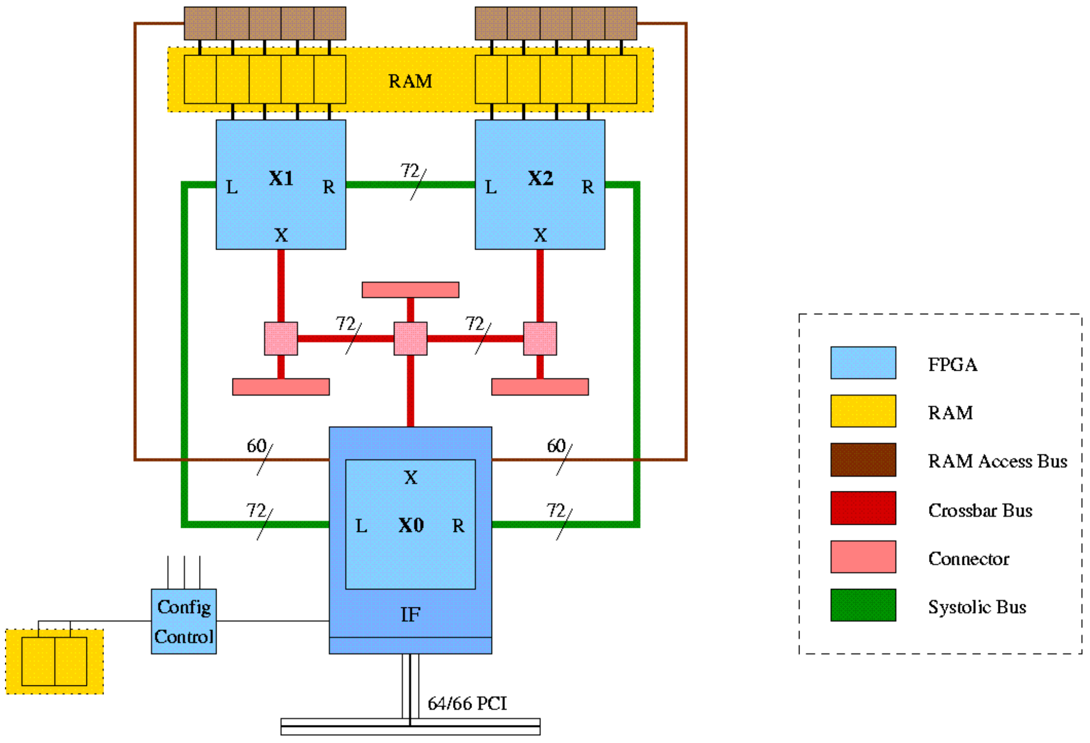


Figure 5.1: *SLAAC1-V block diagram* [47]

The network processor functionality is divided among the primary FPGAs as defined by the structure described in Section 4.1. Given that the target operations of the prototype all involve data movement between wired and wireless networks, this division is made as shown in Figure 5.2. Here, the reconfigurable device *X0* contains the primary system control function, *X1* contains user-based processing functions, and *X2* contains device interfaces along with associated data processing.

All permanent configuration and control functions are embedded in device *X0* since this device has direct access to the Configuration Control FPGA (Figure 5.1). In the prototype system, however, the Configuration Control FPGA consists of nothing more than a data path to the configuration interfaces of the three primary FPGAs. The contents of the *X0* device remain unchanged as users arrive (authenticate) and depart (deactivate). Since it is controlling the configuration operation, the SLAAC1-V design does not allow it to be configured itself.

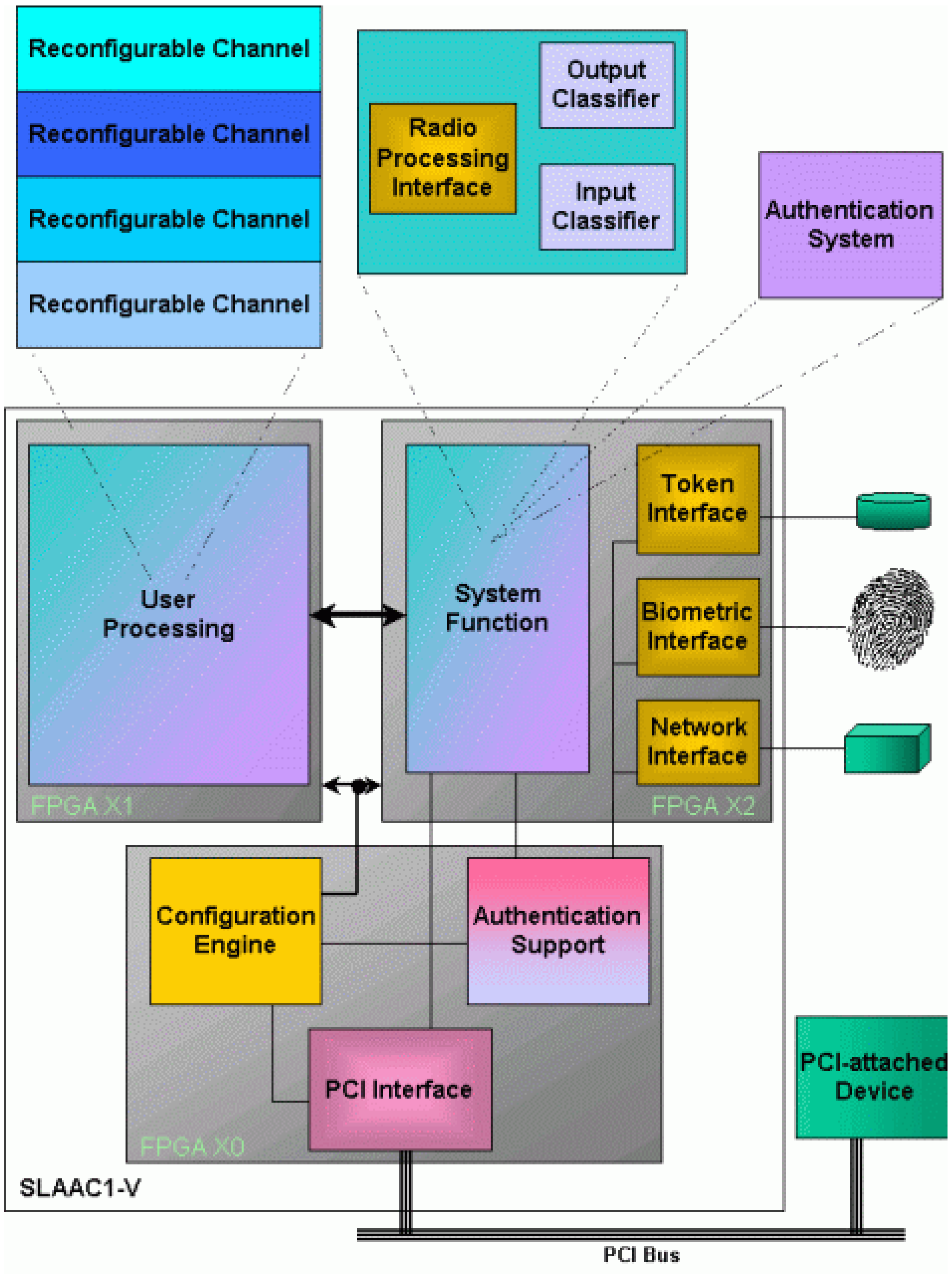


Figure 5.2: *Network processor embedding*

If no valid user is present, device *X2* contains only the User Discovery System and authorization device interfaces. After a valid user arrives, *X2* contains the primary user data routing functions as well as connections to user devices. Add-on modules have been developed to provide this FPGA with physical connections to support hardware, including wired and wireless network, user identification token, and biometric identification device interfaces [47]. These interfaces also perform any packet handling (encapsulation, header processing, error checking) issues associated with an attached device, allowing the data to interact with the user processing system. Additional support for PCI-attached devices is routed through *X0* since it is physically connected to the PCI bus of the prototyping card. As an example, a PCI-attached transceiver is attached to the system described in Section 5.3. This transceiver interacts with the PCI core in *X0*, while user-configurable data processing for the device takes place in *X2*.

Device *X1* is empty when no valid user is present. It is configured to contain the Reconfigurable Processing Channels of the network processor user function depicted in Figure 3.6 when a valid user is detected. This division of functionality allows for run-time reconfiguration of the Reconfigurable Processing Channels in *X1* without disturbing the hardware interfaces contained in *X2*.

5.1.2 Platform Reconfiguration

A Dallas Semiconductor Java iButton [49] is used as the token in this prototype. User discovery and configuration take place as described in Section 4.2. Due to limitations in the token processing power, however, the configuration process does deviate somewhat from the sequence outlined in Figure 4.3. Rather than transferring encrypted configuration stream header information to the token for key assembly, the prototype system transfers the user credentials via a secured link from the token to the network processor platform. The key assembly mechanism is located in FGPA *X0* of the prototype. This modification creates a slightly less secure system, as credentials are exposed to the platform, yet was necessary due to limitations in the iButton token processing capabilities. The FPGA contains more processing resources than the selected token device and provides design access to ease algorithm development. As technology progresses and configuration stream key assembly algorithms are more thoroughly investigated, the assembly function may be moved back into the token.

A user fingerprint template is also downloaded from the token to the platform. This template is forwarded to a Secugen [54] fingerprint reader where a match routine authenticates a user. The result of the match is returned to the network processor as a pass/fail indication. The template is a proprietary format developed by Secugen and cannot be used to reconstruct the fingerprint from which it was derived. The FDA01 device, however, can use it to verify a user using internal hardware to match a current user to the template. It is impractical to perform this algorithm in the FPGA due to its proprietary, undisclosed, nature. In a full implementation of the secure adaptive network processor, however, the hardware to perform this match function should be moved into a token similar to [46].

Upon user verification encrypted configuration data is read from a Configuration Store device. The basis of the encryption used in this system is the Blowfish algorithm [55]. Blowfish was chosen because it supports a flexible key size and maps well to hardware [56]. In addition, the open-source algorithm is similar to other symmetric cryptographic algorithms and is credible in the security industry [57]. Blowfish supports key sizes ranging from 32 bits to 448 bits, although the prototype restricts itself to 32-bit keys to reduce the size and complexity of the embedded hardware. The hardware implementation of blowfish used in the prototype encrypts or decrypts up to 35.5 Mbits/sec (using a 20 MHz clock), regardless of key length. This allows for an entire *X1* or *X2* configuration (approximately 6 Mbits in length) to be decrypted in 170 ms.

Once configured for user operation, the user functions of the current prototype embed wired and wireless network interfaces into the user processing elements. Incoming data is classified as belonging to one or more of four independent processing channels as described in Section 3.2.4. Within each channel, data is processed according to the channel needs before being repackaged for output. An arbiter combines output data from the four processing channels, preserving packet boundaries, and forwards the data to an output device.

5.1.3 Foundation Platform Analysis

In the construction of the foundation platform, steps were taken to produce a reliable system in current hardware. Movement of some token-based processing into the configurable logic of the network processor reduces the security of the prototype itself, but provides an enhanced algorithm development and testing environment. Once development is complete, functions such as biometric analysis and key generation can be moved into the token. An ideal system would use a more powerful token device so that user credentials could remain unexposed during operation and biometric match results are not transferred over an exposed interface.

5.2 Secured DSN Receiver

The first prototype constructed using the secure adaptive network processor structure described above was a receive-only wireless device with a network-attached user display. This system was designed to display data gathered from a Distributed Sensor Network (DSN). Development of this system allowed for basic system concepts to be tested without the need for high data rate services.

A DSN typically consists of *sensor nodes*, *processing elements*, and a *communication network* [58,59]. The sensor nodes gather information and forward it via the communication network to one or more processing elements for analysis. The processing elements may also communicate among themselves via the network. The end result of this interaction is condensed information regarding the sensor results. These results may then be distributed to users in the field. If the DSN is deployed in a battlefield situation, it may be undesirable for a working receiver to fall into enemy hands. In this case, the secure adaptive network processor solution of Chapter 3 is ideally suited to the application.

The SensIT DSN program sponsored by the Defense Advanced Research Projects Agency (DARPA) is an existing battlefield network research system [60]. The version of this system implemented in [60] uses sensor nodes containing several acoustic, infrared, and seismic sensors to gather environmental status information. This information is fed via bi-directional radio links from the sensor nodes to processing elements for analysis and storage in a distributed database system. Stored results include discriminated tracked and wheeled vehicle movement through the sensor field. End-user devices query the database using a Java-based application (Figure 5.3) running on a commercially available Personal Data Assistant or laptop computer. The displayed information provides a real-time picture of battlefield status information to the user.

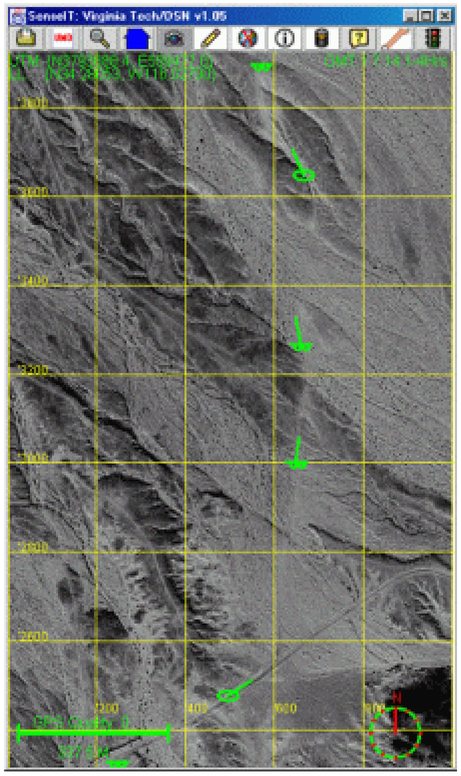


Figure 5.3: *SensIT DSN user interface*

For the purposes of the secure receiver described in this document, all SensIT database results are fed into a single data sink [61]. The sink then broadcasts encrypted versions of the sensor data to operational secure DSN receivers. This broadcast data is divided into three hierarchical classes at the sink. Each class is encrypted independently to allow for a distinction between users at secure receiver devices. Since a Blowfish encryption core was already ported to the prototype to support key management and configuration stream encryption (Section 5.1.2), Blowfish is also used as the data encryption scheme for all classes of sensor network data. Data encryption keys are varied between classes to preserve the security hierarchy of the classes.

The three different classes of information chosen for use in the system are detection data, sensor node location information, and tracking results. Tracking information is available to every user. As such, it is assigned the lowest security level of Class 3. Sensor node information is only provided on a need-to-know basis, and is restricted to Class 2 users and above. Finally, detection information is limited to a select few users categorized as Class 1. Table 5.1 presents this partitioning of user classes as an example of how differential services are utilized in the prototype.

Table 5.1: *DSN user class partition matrix*

User Class	Information Classes		
	Tracking	Sensor Node Info	Detection
1	Available	Available	Available
2	Available	Available	Prohibited
3	Available	Prohibited	Prohibited

An effort to secure the communication among sensor nodes has been made in the SensIT program by Carmen, Kruss, and Matt of Network Associates Technologies [62]. Although it was not done for this prototype, the scheme presented here could also be applied applicable to the sensor nodes themselves. The application of this technology would serve to enhance any secure link between the nodes as well.

5.2.1 DSN Receiver Operational Details

Figure 5.4 is a block diagram of the DSN receiver operational prototype system. The system uses a fingerprint reader and a token to identify a user. Once a user is identified, the system provides a communication path from a wireless receiver to a user display device. A host system supplies configuration information.

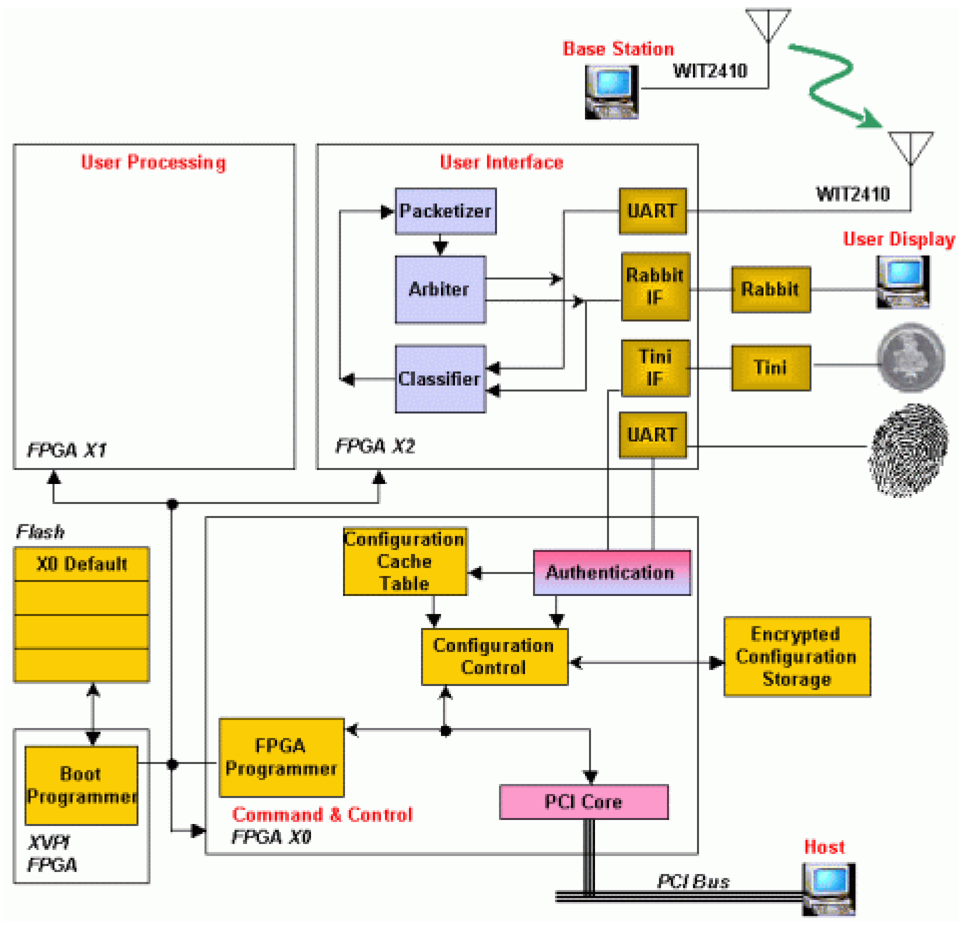


Figure 5.4: *DSN receiver prototype*

In some cases, it may be undesirable to transmit from the DSN receiver device, as this may alert hostile elements and provide a means of tracking a user position. With this in mind, the secure receiver system prototype was designed as a receive-only device. As a result, the use of credentials to obtain a configuration from a remote server is impractical, and the configuration is instead pre-encrypted and stored locally. This varies from the procedure outlined in Chapter 3, as the user configuration is known (and encrypted) beforehand and credential information is not used to construct a key. Once a user is validated at a secure DSN receiver, encryption keys for the configurations used by the current user class are read from the user token and a locally stored radio system configuration is embedded into the secure adaptable network processor.

The operational receiver acquires SensIT data via a radio interface implemented using a Cirronet WIT2410 industrial transceiver [63]. This device operates at 2.4GHz using a direct sequence spread spectrum communications protocol and a proprietary packet format. Multiple WIT2410's communicate in a star topology, where a single device is designated as a base station and all other devices are remotes. In this prototype, a Linux-based server places the SensIT data into the wireless network packets described below in Section 5.2.2. These packets are then forwarded to a WIT2410 transceiver base station attached to the serial port of the server. The transceiver base station encapsulates the transmitted data in its own communication packet and broadcasts the data to all active receivers. The WIT2410 base station to receiver link automatically provides error detection and correction for transceiver packets and the receiver presents only the data contained in valid transceiver packets to the network processor.

Upon reception of data, the network processor strips the wireless network header from the packet and uses it to classify the data into one of three channels for forwarding to the user. Due to time limitations, the decryption units that would have been contained in the reconfigurable channels of FPGA X2 were not implemented. Instead, channel data is simply placed into three outgoing FIFO queues within FPGA X1. This data is read from the queues by a Rabbit Semiconductor RabbitCore RCM2100 module [64] as described in [47]. The Rabbit embeds the data in a UDP packet and broadcasts it from a wired 10-baseT Ethernet network port for reception by the user display device.

The user display device is attached to the Rabbit Ethernet port via a crossover cable. This device runs an application that removes the data from the UDP packets and builds a local version of the SensIT database. This database can then be queried using the front-end depicted in Figure 5.3.

5.2.2 Wireless Network Data Interface

Data destined for the transmission from the wireless interface is packetized within its processing channel before being presented to an arbiter (Appendix E) for multiplexing onto the wireless output channel. This encapsulation of data into a wireless packet is performed by the *packetizer* entity provided in Appendix D. This entity adds an appropriate wireless protocol header to outgoing data and calculates an attached CRC if it is needed. The wireless packet format is shown in Figure 5.5.

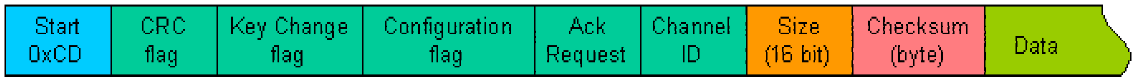


Figure 5.5: *Wireless packet header*

A *Start* marker in the wireless packet header allows for receiver synchronization and a *Checksum* [65] is provided to verify both header integrity and the validity of the Start marker. A control byte, identical to that used on the wired network input is included to provide packet content and routing information. Also in keeping with the wired network input format, the *Size* word is presented LSB-first. If a CRC is present, it is appended to the data and is calculated over the full packet beginning with the Start marker.

This wireless packet format was constructed to minimize header overhead while allowing for lossy connections between wireless stations. As indicated above, the wireless interface for the first prototype was implemented using a commercial wireless transceiver. This device provides internal data integrity checking for its own smaller communication packets. Packets seen by the prototype at the wireless interface consist of several of these smaller transceiver packets. Data errors at the wireless interface therefore consist of either a lost portion of a packet or reception of data beginning in the middle of a packet. Lost portions are readily identified by the CRC, while the checksum provides some assurance that a detected packet start marker is indeed the start of a packet. The checksum and CRC employed are the same as those used in the Ethernet protocol, and should provide similar data integrity assurance. The chosen Ethernet techniques were primarily selected for ease of hardware implementation, and have been carried over to the second prototype (Section 5.3) in which a user-configurable wireless system is used [66]. The data integrity checks provided in the current system may very well be replaced as the wireless interface continues to evolve, and new data transfer protocols are developed.

Data received by the wireless transceiver is classified for channel processing by the *classifier* entity of Appendix F. This entity routes data to the correct processing channel, performs any needed CRC check, and signals an acknowledgement unit when the sender requests an acknowledgement of packet receipt. In this case, the classifier also acts to detect valid packet starting points by searching for a packet start marker and verifying the header checksum. Wireless packet information is reduced to the control byte as data is forwarded to a channel for processing.

Data flowing from the wireless to the wired interface is placed into outgoing FIFOs after channel-specific processing is complete. A separate Control FIFO is used to indicate the availability of data in these outgoing data FIFOs. This data is retrieved by an outbound Ethernet interface for placement on the wired network.

5.2.3 Wired Network Data Interface

A header is required for all data entering the wired network side of the processor, as shown in Figure 5.6. This header allows for classification and processing of packetized data before it is sent to the wireless interface for transmission. The *Control Byte* determines the processing channel to which the packet will be routed as well as providing some packet processing control information. The *Size* fields specify the size of the data set in bytes, where the size is a 16-bit value provided Least Significant Byte (LSB) first. A network

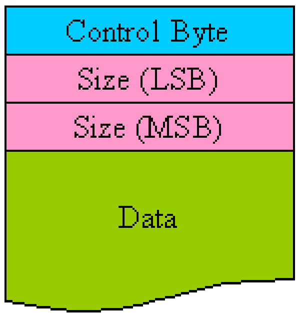
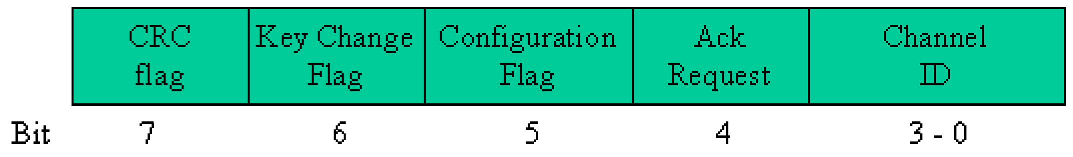


Figure 5.6: *Wired network input packet*

protocol-specific module translates from the external network packet format to this internal format.

The *user_classifier* entity provided in Appendix C parses the control byte (Figure 5.7) to determine packet destination. An optional 16-bit Cyclic Redundancy Check (CRC-CCITT [67]) may be attached to the data for error checking. If that is the case, the *CRC* flag is set in the Control Byte, the Size field is expanded to include the CRC, and the user_classifier performs a CRC check over the data set as it is read in. The result of the CRC check is passed to the appropriate channel following the data. The *Key Change* flag in the Control Byte is set to indicate the fact that the data in the packet is a new key to be used for following transmissions on the channel. The *Configuration* flag indicates the packet data is a new configuration for the indicated processing channel. An *Ack Request* field allows the sender to request an acknowledgement of packet receipt from the recipient. If an acknowledgement is requested, the user_classifier signals a separate acknowledgement unit to reply. Finally, the *Channel ID* field provides a mask for routing this packet, where each bit is used to enable a processing channel. Note that data may be broadcast to more than one channel by setting multiple bits of the Channel ID.



	CRC flag	Key Change Flag	Configuration Flag	Ack Request	Channel ID
Bit	7	6	5	4	3 - 0

Figure 5.7: *Incoming data control byte*

The Rabbit Interface gathers raw data that is to be transmitted from the wired interface from four internal FIFOs, each representing a processing channel. The data is then encapsulated in an appropriate UDP packet by the Rabbit device and sent to the attached user display device.

5.2.4 DSN Receiver System Analysis

As a prototype, this system demonstrates the basic functionality of the network processor. It does not, however, implement the full security scheme, and contains several weak links in its device protection. The reliance on an external module for user validation is clearly the weakest point of the system. The returned pass/fail response could easily be intercepted or generated by a malicious user. A better solution would be to secure the link between the biometric device and the FPGA. Hardware available at the time of prototype construction did not allow for a secure link to the biometric device. However, it is very likely that such secured authentication hardware will appear in the near future. In fact, with a small hardware investment, a device manufacturer could easily implement the scheme used to communicate with the token to enhance the security of the match response.

The use of local configuration stream storage simplifies the design and reduces configuration time. However, anyone acquiring the device will obtain encrypted versions of the various user configurations as well. This is also a weak point in system security, since the encrypted configuration data may be leisurely analyzed and a clear configuration might eventually be produced. In a non-prototype system, using the secured remote storage scheme described in Section 3.2.3 would enhance device security by eliminating this risk. However, as mentioned earlier, there are cases in which it may be undesirable to transmit from the device at any time. In addition, the security enhancement could result in unreliable operation and slow startup times as the configuration data is retrieved from a wireless interface. To address these issues, two options are available for production systems: the base station could intermittently broadcast configurations (like the Configuration Store, but without waiting for requests) that can be used as needed, or it could be assumed that the devices will be lost and encryption channels that modify keys for data reception (rather than full device configuration) during user operation could be included. The first option would reduce available bandwidth for data transmission; the second would protect the data while still leaving the underlying technology exposed.

It is worth noting that this prototype system does not make use of all features of the network processor. Configuration is limited to locally stored bitstreams and the reconfigurable processing channels are essentially null devices. Furthermore, once a receiver has been activated by a user, no further run-time reconfiguration is done. As mentioned above, additional features in each of these areas could be added to the system to improve its security or enhance its functionality. Time and resource constraints prevented this from being done for this particular prototype. Instead, as additional features were added the system evolved into the second prototype discussed in Section 5.2. For further analysis of the Secure DSN receiver system, including throughput and latency characteristics, see Abraham [47].

5.3 Secured Network Gateway

Having laid the groundwork for a user application with the DSN receiver system described in Section 5.2, a second-generation prototype was constructed within the network processor foundation of Section 5.1. This second system is essentially an enhancement of the Secure DSN Receiver, as it builds upon the packet handling and authorization/configuration interfaces of that system. Making use of alternate hardware for both the network and radio connections, the new system increases potential throughput. In addition, the user processing space of *X1* is used for reconfigurable processing channels that are used to encrypt and decrypt data traffic for the wireless network. Finally, more of the security features described in Chapters 3 and 4 are included as configuration data is retrieved from a remote storage device on the wired network side of the system.

The user function provided by this second prototype is a Secured Network Gateway. A Secured Network Gateway is a device providing a link between two network systems that is adapted to a particular user and active only in that user’s presence. Applications for this type of device include any high-security installations with a private network that have a controlled external access point. This type of installation might include the radio room of a naval vessel or the communications center of a military installation. While many users may not need this level of security, the adaptable nature of the system does provide some benefit even the security features are not essential. For example, simply indicating the presence of a particular user may modify the features of a gateway. Real benefit for all users can be seen in the adaptable nature of the system as it is deployed in devices like cellular telephones where the interface between voice and radio may be adapted to suit the services used by an individual. In this situation, run-time adaptability of the system also comes in to play as the user moves from one location to another, potentially changing radio communication protocols along the way. This section describes a Secured Network Gateway prototype targeted toward the radio room of a naval vessel. It requires the presence of a valid user to provide a link between a wired network and a wireless system. A block diagram of the system is shown in Figure 5.8.

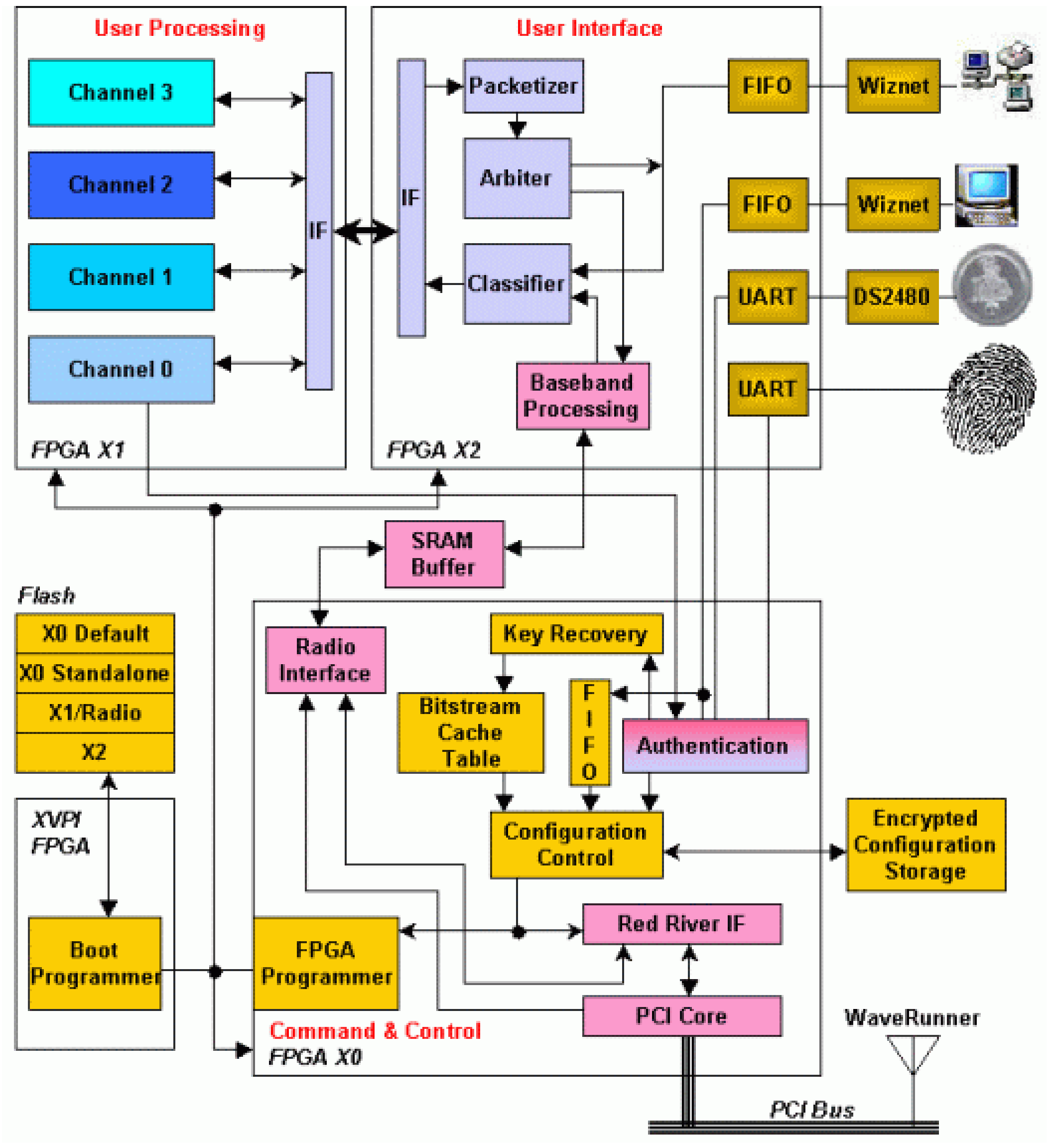


Figure 5.8: *Secure network gateway prototype*

5.3.1 Network Gateway Structure

Configuration of the gateway is done in a secure manner using a remote Configuration Store. Once a user is authenticated, the user requirements and credentials are transferred from the token to the network processor. The network processor then initiates a configuration request sequence to retrieve user settings for the *X1* and *X2* FPGAs as well as the connected radio hardware. These requests and the return data are handled as described in Section 3.2.3, with the exception that the information contained within the Configuration Store is not pre-encrypted and does not require a domain configuration key for decoding. While the addition of this security level could easily be added, it is more convenient to deal with unencrypted bitstreams in the prototype environment. The encrypted user configurations stored in local SRAM are therefore encrypted only with the one-time key used for the transfer. A local security agent processes the attached header data to create keys for each configuration and writes the results to the Bitstream Cache Table. The Configuration Manager (Section 4.3.2) then parses the table and configures attached devices as needed.

The flow of data through the network processor portion of the system follows the scheme outlined in Section 5.1.1. The wireless network still operates on a serialized FIFO-based interface with an arbiter feeding data to it and a classifier reading from it. The wired network still feeds data to a classifier, but it now uses an output arbitration unit to feed multi-channel data through a single path. A second wired connection is included as well to allow for a directly connected user device (PDA, laptop PC) for control and monitoring in addition to a local area network (LAN) connection for data flow.

The updated wireless interface is provided by a Red River Engineering WaveRunner 304 configurable radio platform [66]. This device allows for user-specific baseband protocol processing. An external 2.05GHz radio frequency upconverter is used to move the intermediate frequency produced by the WaveRunner up to a fixed broadcast frequency. A fixed-frequency downconverter performs the same function for received data.

Wiznet IIM7010 Ethernet modules [68,69] are used for the wired network interfaces. These modules read and write data using a FIFO interface and provide raw Ethernet, UDP/IP (User Datagram Protocol over IP), or TCP/IP network protocol handling. For this prototype, data exchange with the LAN is done at the Ethernet level, while data is exchanged with the local user device using TCP/IP via a crossover cable connection. The Configuration Store is contained within the user device to simplify testing and demonstration.

5.3.2 Secure Network Gateway Analysis

When compared with the original secure DSN receiver prototype, this system better demonstrates the capabilities of the secure adaptable network processor. Here, several resources are reconfigured based on the presence of an authorized user, including internal network processor resources and an external radio device. All of the configuration data is retrieved in a secure manner via the packetization scheme described in Section 3.2.3.

This prototype system is still under construction, and is scheduled to be ready in 2003. At the current time, the SLAAC1V platform system is still contained within a host system and the configuration retrieval security system has not been put in place. When it is complete, this prototype system will demonstrate the proposed security scheme and be self-contained in a portable system. This portable prototype will form a portion of the demonstrable result of the NAVCIITI research effort.

5.4 Summary of Experiments

The prototype systems presented in this chapter provided an example of the capabilities of the secure network processor. These examples demonstrated how the network processor might be implemented, and highlighted its strengths and weaknesses. The examples also showcased its versatility, using it both as an edge device and within a network system for providing secure user-based operation of network systems.

From these experiments, it is clear that the secure adaptable network processor has a place in future communications devices. It is also clear that additional support for secure external systems (tokens, biometric devices) needs to become readily available before the full benefits of the secure processor will be realized. By implementing the system in standard reconfigurable logic, these prototypes show that the device itself can be constructed using current technology. Minor modifications to this technology, including built-in support for random number generation and other cryptographic functions would allow the technology to be more easily embedded in a wide variety of end-user devices.

CHAPTER 6

"He who esteems trifles for themselves is a trifler; he who esteems them for the conclusions to be drawn from them, or the advantage to which they can be put, is a philosopher."
- Edward Bulwer-Lytton

THE RESULTS AND ANALYSIS

Several approaches to the design of a network processor currently exist, each of which provides some means of efficiently handling high-speed network traffic and adapting to new protocols. As these devices find their way into applications in which the very algorithms need to be protected, it becomes more important to conceal the device function, control user access, and otherwise limit the possibility of interacting with, participating in, or interfering with the operation of a privileged communications system. Most design techniques currently used to build these network processors do not, however, attempt to conceal the internal algorithms or to provide support for security of the hardware itself. Those that do provide some security support typically add hardware co-processing for common security functions (RSA, SHA, etc.) or provide internal storage for program streams. While these solutions are steps toward system security support, they still fall short in device protection as external connection monitoring, scalability, and analysis of both power and timing come in to the picture. When the reputation of a network company is based upon its ability to process data more efficiently than the competition, it is important to protect the intellectual property of a design. When that design is critical to the security of the user, it is important that device security not be breached.

This dissertation has presented a network processing system that addresses these issues of device-level security. The concept of a secure adaptable network processor was introduced in Chapter 3, and a structure to support that concept was defined in Chapter 4.

Chapter 5 established the feasibility of constructing devices based on this structure by describing a working prototype platform for the defined processor structure upon which two distinct user applications were embedded. The resulting systems demonstrated the potential of this device both as a security system and as a basis for user adaptation of a running system. This chapter summarizes the findings of the previous chapters and provides insight into future directions for the work.

Section 6.1 provides a high-level analysis of the design response to various board-level attacks and outlines how the modifications made in the two prototype systems affects their resistance to these attacks. Section 6.2 reexamines the objectives that were put forth in Table 3.1 for the creation of a system that is resistant to board-level attacks and specifies the system properties that define this secure adaptive network processor. Section 6.3 lists the contributions made by this work and identifies some directions that additional research into this topic might take.

6.1 Security Analysis

For the purpose of this analysis, the system is divided into three primary components as shown in Figure 6.1: the network processor itself, the external authorization hardware, and the external Configuration Store. Each of these components can be replaced in operation and the interfaces between them can be monitored or replaced as well. The eight following scenarios can each occur in an operational system that is under attack. In each scenario, Alice (A) and Bob (B) will be assumed to be owners of valid devices, while Mallory (M) will be the attacker. These attempts to discover the internal function of the secure network

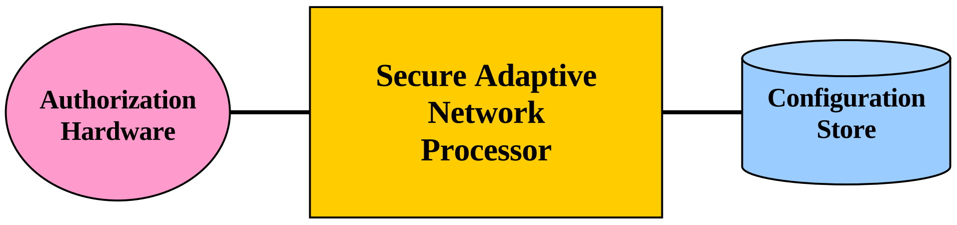


Figure 6.1: *Basic security analysis block diagram*

processor are considered beginning with the authorization hardware and moving to the right through the system as summarized in Table 6.1.

Table 6.1: *Attack Consideration Matrix*

Considered Attacks	Description
Authorization Replacement	The attacker substitutes non-standard entities for the external token and biometric device.
Authorization Platform Replacement	An attacker-controlled platform is substituted for the secure network processor during user authorization.
Authorization Man-in-the-middle	The direct link between the external authorization devices (token, biometric) is broken and two links to the attacker are substituted.
Authorization Interface Snooping	The attacker listens to all transactions between the external authorization devices and the network processor while a valid user is authenticated.
Configuration Store Replacement	An attacker-controlled device replaces the Configuration Store.
Configuration Platform Replacement	Configuration request messages are sent from an attacking device to the Configuration Store.
Configuration Man-in-the-middle	The direct link between the network processor and the Configuration Store is broken and two links to the attacker are substituted.
Configuration Interface Snooping	The attacker listens to all transactions between the network processor and the Configuration Store during a valid configuration request.
Operational Bus Monitoring	External data busses are monitored while the device is operating with a user function.
Interface Flooding	Device authentication and configuration interfaces are flooded with attacker-generated data.
Power Attacks	Differential power usage is analyzed in an attempt to discover encryption keys.

6.1.1 Authorization Replacement Attack

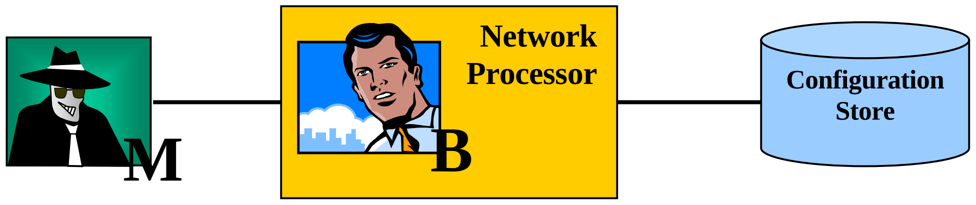


Figure 6.2: *Authorization replacement attack*

In this attack, Mallory (M) will replace the authorization device (token) with a false system in an attempt to use Bob's (B) secure adaptive network processor in an unauthorized fashion. It is assumed that Mallory knows the basic electrical structure and protocol used for communication on the interface. When Mallory arrives at the system, Bob's device will encrypt a one-time random number and send it to Mallory's token replacement. In order to return the correct response to initiate a connection, Mallory will then need to know the encryption scheme and a valid token key. In other words, this operates in the fashion of a standard challenge-response protocol.

While the encryption scheme may, in fact, be a matter of public knowledge, the private key portion of the public/private key system used for data encryption between Alice and Bob is a secret kept in Alice's token. Mallory may obtain a valid key if Alice's token can be compromised by offline inspection or an online attack like those described in Sections 6.1.2, 6.1.3, and 6.1.4. With a key length of 128 bits, there are 2^{128} possible keys. On average, half of these must be tried before the correct key is discovered. With the authorization process taking several seconds per attempt, this would mean that Mallory would typically need to spend more than 5.4×10^{30} years inserting a token and feeding in a biometric result before succeeding. In other words, the platform would long be obsolete and the known universe would be gone before it was broken into in this manner.

Assuming a connection between Mallory's token replacement system and the platform can be established, a correct response from the biometric device is also required. The biometric information contained in a prototype token is a template based upon a fingerprint. It would be necessary to know (or be able to reproduce, e.g. with similar commercial hardware) the template generation algorithm in order to create a valid alternate biometric identifier. In this aspect, the prototype system falls short since obscuring the algorithms and data structures used for production of fingerprint templates provides the sole protection for data exchanged with the attached biometric device. The biometric device used is commercially available, and a second device could be used to produce alternate fingerprint templates for inclusion in Mallory's token replacement. Furthermore, the device attached to the prototype platform does not encrypt requests or responses. While reading a user fingerprint template that is being sent to the matching device would not reveal the user's actual fingerprint, it would provide information that is contained in the token interface transfer, and could be used to assist in decoding other transferred information. The commercial fingerprint device also returns a clear match response when validating a user. An attacker could easily reproduce this response. For the biometric to be a truly meaningful part of the system, an attached matching device needs to provide a secure interface (the scheme used for the token to platform interface would suffice). A better solution would be to use the biometric scanner only to produce data, secure the link between the scanner and network processor (in a fashion similar to that used for the token/platform link), and perform the matching algorithm in the network processor. Unfortunately, the proprietary nature of matching algorithms makes this unlikely in a prototype system.

Since authorization interface traffic is encrypted using a one-time-use random number (nonce) for each transaction, observation of a prior valid user interaction will not return information that can be used to determine the correct responses to network processor queries. This interface is similar to a public key / private key exchange, except the public key is kept privately in a device and never exposed on an external interface. The security provided for data flowing on the interface is on the same order as the chosen public / private key scheme. In addition to providing the correct responses, Mallory will need to provide valid tags for data that is to be retrieved from the Configuration Store along with the credential information and hash function needed to reconstruct the key used for the retrieval transaction. Since the scheme proposed in Chapter 3 never moves the credential information out of the token, this would imply that Mallory must know the contents of a valid user token, since the data tags and credential information are secrets shared only with the Configuration Store. If insecure tokens are used this is certainly possible, as they are as susceptible to physical examination as any ASIC or permanent memory-based device. In the ideal system, the security of individual token data is dependent upon the physical security of the token device itself.

With these points in mind, it is clear that this attack will succeed only if Mallory is, in fact, a valid user or knows enough about the system to create valid tokens. Should either of these cases hold true, Mallory could cause the network processor to be configured with something from the Configuration Store. This would allow for the examination of that IP while Mallory's authorization replacement remained in place. In this case, that particular IP would be compromised. It is worth noting, however, that only the IP that Mallory knows the tag and corresponding credential information for is compromised. If the attack information came from a compromised token, only the IP accessed by that token is exposed.

The use of maintenance keys with periodic updates helps to alleviate the likelihood of this type of attack by expiring tokens. If the content of an older token is compromised, it may no longer produce a working system, as the keys might have expired. In addition, the token devices used for the prototype systems claim to erase internal contents when opened. Physical examination of the internal system would therefore produce no useful information.

6.1.2 Authorization Platform Replacement Attack

In an effort to gain the information needed for the authorization replacement attack described in the previous section, Mallory might attempt to discover the contents of Alice's (A) token by replacing the secure network processor with an alternate device. To be successful in this endeavor, Mallory must generate a number of the correct length and send it to Alice. Alice's token will run a decryption algorithm on the provided information to produce another number, then use that number to encrypt the data that is returned to Mallory. The proper channel encryption scheme does not need to be known for this process to generate a response from Alice's token; however, Mallory must know the encryption scheme, packetization technique, and the correct secret platform key to extract useful information from the resulting packet. In other words, without intimate knowledge of the user discovery system, no more information is provided by this attack than is available by monitoring the interface during a normal authorization cycle as described in Section 6.1.4. If the discovery application of the secure adaptive network processor were fully reverse-engineered prior to this exchange, however, Mallory may know both the details of the exchange process and a valid platform key, and thus possess enough information to decode the response provided by Alice. As a result, Alice's token would be compromised.

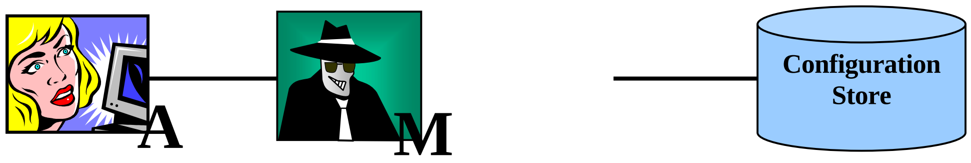


Figure 6.3: *Authorization platform replacement attack*

Efforts to prevent this attack must therefore be focused on producing a discovery application that is not prone to reverse engineering and limiting the transfer of confidential user information from the token to the network processor. The proposed scheme addresses the second of these issues by keeping all user credentials in the token device. If Mallory fully duplicates the user authentication mechanism and Alice can be convinced to authenticate at Mallory's system, Mallory will be able to observe Alice's user function, but will not gain possession of her credentials.

The first step toward addressing the issue of protecting the user discovery application itself is to encrypt bootstrap information so that the structure of the discovery system is not evident from stored data. Of course, this data must be decrypted prior to initial configuration of the network processor. The use of a device with a fixed built-in security key allows for encrypted bootstrap data, but leave the network processor device internals susceptible to offline examination. Similarly, including any sort of permanent fixed hardware function in the network processor, whether it is a secure hash of the processor content or a subset of the authorization scheme, would result in a user discovery application that is no more secure than an ASIC. The use of a soft key similar to the current Xilinx *Virtex-II* FPGA configuration encryption scheme for bootstrap configuration will negate the effectiveness of this type of attack. This key can be varied between network processor devices during system assembly and is lost when power is removed from the device. As a result, a wide variety of bootstrap keys can be employed without the risk of ASIC-like reverse engineering. Since this particular prototype system was built upon a platform that does not support bootstrap encryption, they are susceptible to this type of attack. Moving the prototype onto a *Virtex-II* device (see Section 4.1) will allow for a secure bootstrap.

If increased security of the bootstrap operation were required, a somewhat more secure version of a discovery application would be one that simply recognizes a token insertion, and then requests a complete authorization application from the Configuration Store. The full authorization algorithm would include the structures necessary to set up the secure link and authorize users. It would arrive at the system encrypted like any user application. A portion of the key for this configuration can be built into the lightweight discovery application or network processor hardware and the token can provide another part of the key. If a secure division of the key is used, neither the loss of a token nor the examination of the initial user discovery application will provide enough information to decode the full authentication system.

6.1.3 Authorization Man-in-the-middle Attack

The Man-in-the-middle attack is not a particularly productive attack on the system. In this attack, Mallory is intent on acting like a proper platform for Alice and a proper token for Bob. The attack begins by Mallory pretending to be Bob as Alice begins authorization. Mallory then uses information supplied by Alice to pretend to be Alice for Bob's user discovery application. The intent is either to modify information passing between the two of them during authorization or to gather information from one side to assist in providing correct responses to the other side at a later date.

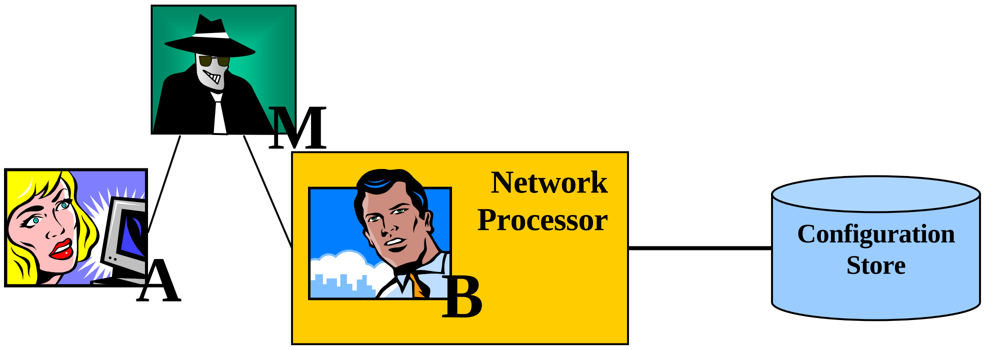


Figure 6.4: *Authorization man-in-the-middle attack*

Simple relaying of information during the attack is no more useful than a simple interface snooping attack (described in the next section). To make this attack worthwhile, Mallory needs to be able to modify data and generate proper responses for both sides. In other words, both of the attacks described in Sections 6.1.1 and 6.1.2 need to be successful. If correctly inserted, Mallory could cause the platform used by Alice to contain an incorrect application by modifying the requested application tags passed to the platform. Any inserted application tag would, of course, need to identify a valid application contained in the Configuration Store in order to get a response from that device.

6.1.4 Authorization Interface Snooping Attack

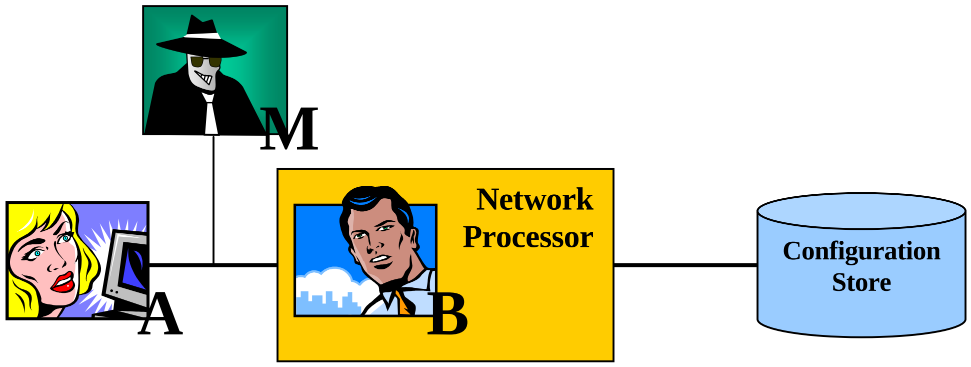


Figure 6.5: *Authorization snooping attack*

The purpose of this attack is to either gather sensitive information directly from data flowing on the authorization interfaces (credential information, user biometric data, etc.) or to collect the data for a later authorization replacement attack in which the data is simply replayed with the goal of obtaining a user configuration without the user being present. The success of this type of attack is based upon the likelihood of a journal or dictionary attack succeeding on the interface.

The authorization interface is protected from bus snooping attacks by the security scheme used in the establishment of the processor/token link. Encryption of data flowing on this link provides a measure of privacy similar to a public key / private key scheme for the transferred information, including user configuration identifiers, biometric template information, and working keys for user configuration data. Public and private keys are kept in the secure network processor and token, respectively. Additional security is provided by the use of a nonce in the construction of the keys used for any particular session. This encryption hinders attacks aimed at directly gathering sensitive information from the data flow. Replay attacks are thwarted by the use of a nonce as the basis of the link establishment. This value will change with each authorization, resulting in a different response for the same user. Furthermore, while a brute force attack on logged data could reveal the key for a particular session, the amount of time required to do this makes the attack impractical within the span of a single session and other sessions will use a different key. A brute force attack tries all keys until the correct one is found. With a key length of 128 bits, there are 2^{128} possible keys. On average, half of there must be tried before the correct key is discovered. While offline analysis can be done significantly faster that the brute force attempt at authorization replacement, if a teraflop machine could try one key per operation it would still take 4.9×10^{18} years to find the correct key.

In an ideal system, compromising this link while the system is operating would not reveal any critical IP or user-specific data. In the prototype systems, however, the token does not possess sufficient processing power to generate working keys. The prototype systems therefore transfer credential information from the token to the platform. If Mallory is able to decode this information from logged sessions, credential information could be lost. Although extremely unlikely (see the previous paragraph), a successful attack would be a significant step toward producing unauthorized tokens.

6.1.5 Configuration Store Replacement Attack

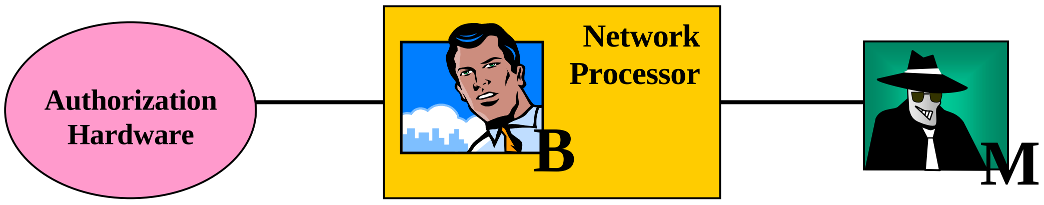


Figure 6.6: *Configuration store replacement attack*

In this attack, it is assumed that a valid user has been authorized at Bob's secure adaptive network processor, and a request for configuration data has been generated. Mallory is attempting to configure the network processor with something other than the authorized user function by replacing the authorized Configuration Store.

Upon receipt of the request, Mallory must decrypt it (if it is encrypted) to determine the requested configuration stream tag. The difficulty of this operation depends on the strength of the cryptographic algorithm used. For most modern algorithms, the time required to accomplish this is on the order of the brute force attack described in the previous section. Once the request is decrypted, the credentials associated with that tag must be used to produce a valid one-time working key for a configuration stream, and the replacement stream encrypted using this key. Prior to this packet encryption, Mallory must either possess or create a configuration bitstream that is correctly encrypted for the domain of the request. If all of this can be done, the configuration data packet can then be returned to Bob and he will use it to embed a malicious user function.

Success of this attack requires intimate knowledge of tag / credential association, the hash function for one-time key construction, the packet format, and the shared secret credentials themselves. It also requires a valid domain configuration key for the domain from which the request originates. This information cannot be deduced from the authorization or configuration interfaces, since credential information is never communicated over these links. Furthermore, it cannot be derived from examination of the user discovery application running on the network processor since it does not posses the credential information or hash function used in key construction. Finally, while the Configuration Store does contains sufficient information to produce a valid packet, it does not have access to the domain configuration key set used to pre-encrypt the configuration data. In other words, more than one component of the system (the Configuration Store and a valid token from the domain under attack) must be compromised for this attack to succeed.

Configuration data flowing in this system is self-authenticating. Only an attacker with intimate knowledge of the Configuration Store could produce a valid packet encrypted with a valid credential-based one-time key for the current user domain that would be examined and accepted by the network processor. Additionally, this data will be discarded as invalid if it is not encrypted with the correct domain configuration key. An attack could succeed, however, if Mallory managed to take over the valid Configuration Store (containing all of the algorithms and credential information required to produce the returned data) and either reverse engineer a valid token for the requesting domain or obtain the domain configuration key through some other means, such as compromising the token generation system. Fully compromising this part of the system would result in a complete loss of all system security for the given domain, as all credentials and the domain key would be known; therefore, much depends upon the integrity and physical security of the tokens and token generation system.

6.1.6 Configuration Platform Replacement Attack

Looking at the system from the other side, Mallory might attempt to pretend to be a secure network processor device so that Alice's Configuration Store returns configuration information. In this case, Mallory would be attempting to determine stored IP by examining configuration data.

The attack would begin by Mallory sending a configuration request to Alice. Of course, the request must be properly formatted, encrypted with a valid key, and refer to a valid secret configuration tag. While the formatting and encryption scheme might be deduced by examining a running user discovery system (see Section 6.1.2), the tag information would need to come from a valid user token. Sequential or random tag numbers attached to several requests could easily be caught at the Configuration Store when a match does not occur and all further requests from that particular network processor can be denied.

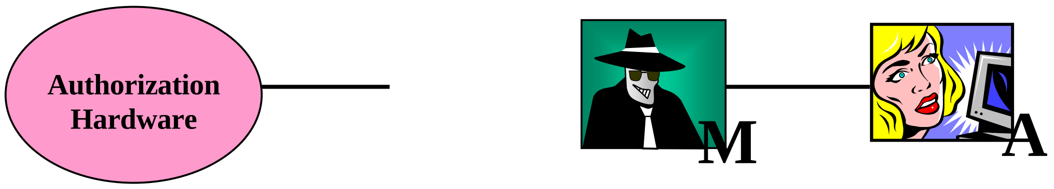


Figure 6.7: *Configuration platform replacement attack*

Assuming Mallory sends a valid request with an acceptable tag, Alice will return an encrypted configuration stream. At this point, however, Mallory encounters the same problem that surfaced in the attempt to replace the Configuration Store. It is essential to know the credentials associated with the returned information, the working key construction technique, and the domain configuration key to extract the configuration information from the returned data. A brute force attack may be able to decipher the configuration packet contents and then the encrypted configuration data, but encryption based upon common techniques (e.g. Blowfish) with reasonable key lengths make this impractical. Should Mallory have access to the contents of a valid user token, however, the key construction technique, credentials, and domain key would be known. In this case, the IP associated with that user could be decoded.

For this attack to be successful, Mallory will need to compromise two system components. Reverse engineering of both a valid token and the user discovery application embedded in a working network processor will provide enough information to retrieve a single user's configuration data. In the prototype systems, Mallory's task is simplified, as the bootstrap user discovery application is not encrypted. Moving to a *Virtex-II* basis for the platform will protect the user discovery application. In addition, the prototype system places key construction hardware and credentials in the network processor upon authorization of a valid user. By modifying and replacing the user discovery application, then enticing a valid user to attempt authentication, Mallory could gain the token information needed for this attack. Once again, a lack of processing power in the token results in reduced security of the prototype. However, as tokens become more powerful, key construction techniques developed on the reconfigurable testbed can easily be moved back off of the network processor and on to the token.

This attack is unlikely to succeed both in the construction of a valid request and in the ability to use any returned information. Should it succeed due to loss of token data and compromise of the user discovery application, however, only the IP for the user whose token was compromised is available to the attacker. Other user functions would not be sent to the system and the breach (if discovered) could be somewhat contained by expiring a single user token or domain configuration key.

6.1.7 Configuration Man-in-the-middle Attack

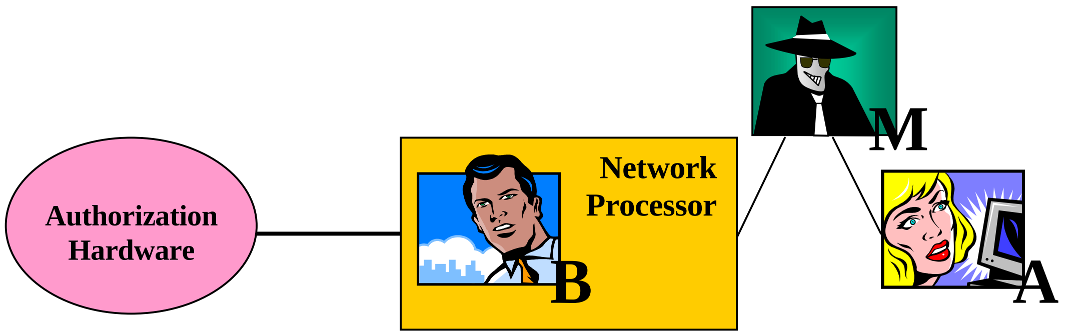


Figure 6.8: *Configuration man-in-the-middle attack*

Like the authorization Man-in-the-middle attack of Section 6.1.3, this attack is the least likely of the Configuration Store attacks. In a successful attack Mallory could change request information sent to Alice or modify the configuration information returned to Bob. The result would be that Bob's secure network processor would configure with the wrong application. If Mallory should chose to modify the configuration information, it would be possible to place an application into Bob's processor that fed protected data out an unprotected channel or modified his system operation in some other undesirable fashion.

For this attack to succeed, the full details of the communication scheme, including all secret credentials, data tags, and domain keys used by Bob, must be known by Mallory. In essence, this attack can only succeed if Mallory could replace both Alice and Bob as in the attacks described in Sections 6.1.5 and 6.1.6, and the overall effect on system operation will be as described for those attacks.

6.1.8 Configuration Interface Snooping Attack

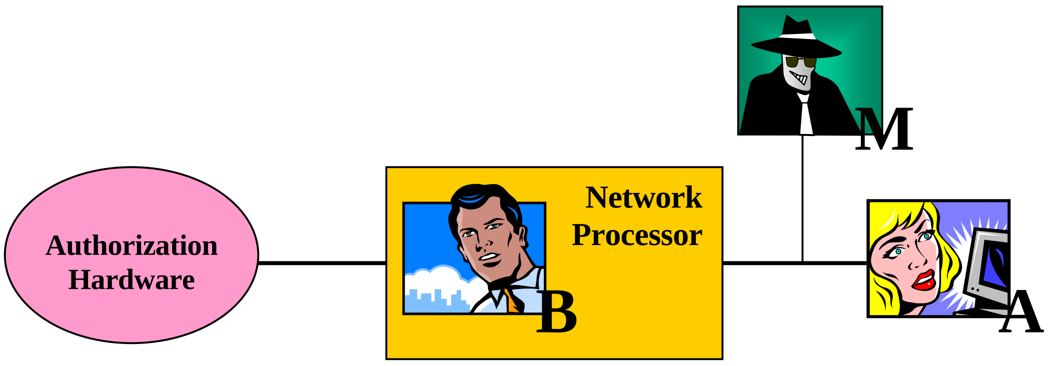


Figure 6.9: *Configuration snooping attack*

Monitoring the exchanges between Alice and Bob will provide access to all information transferred between them. Unfortunately for Mallory, all of packet information is encrypted with one-time keys and configuration information is further encrypted with symmetric domain keys. If these encryptions are done with an accepted protocol like the Blowfish algorithm used in the prototype systems, data flowing on the link will inherit the security of that protocol. For example, at the present time it is suspected that Blowfish may be have some susceptibility to dictionary attacks [55] and attacks that can break or exploit weaknesses in limited versions of the algorithm are known [70].

In systems like the first prototype, where configuration information is permanently stored locally at the network processor, Mallory might have more success. Here, the success of a brute force attack would produce the key (after an average 4.9×10^{18} years – see Section 6.1.4) used for a particular user function. Since this function does not change, any user requesting it would invoke an application for which Mallory knows the operational details.

6.1.9 Other Attacks

Three additional attacks might be considered that are not discussed above. They are operational bus monitoring, differential power analysis, and flooding attacks. In the first case, Mallory observes data busses after the user function has been loaded. In a microprocessor-based system this might result in loss of IP as the instruction stream is read in to the system. In this device, there is no external instruction stream, so the attack is not effective. The second attack relies upon observation of power fluctuations as key components are calculated. As discussed in Section 3.1, this type of attack is not effective for hardware-based encryption like that used in this device. The last attack involves the generation of excessive data on a port in an attempt to deny (or gain) access to the network processor. The security protocols used on the configuration and authorization interfaces assure that this attack will not provide access to the device, as there are no buffers to overflow and invalid data is simply discarded. Flooding either interface would result in a denial (or reduction) of service, however, since valid data would not be able to get through. Flooding of the authentication interface would involve adding something to a local token connection, and is not likely in the presence of a user. Flooding of a network-attached configuration interface is more likely to produce a reduction in service, but will most likely not result in a loss of service altogether, and is not dealt with here.

6.1.10 Attack Analysis Summary

The security of the proposed adaptable network processor essentially hinges on the security of two interfaces – authorization and configuration. Authorization takes place using a challenge-response mechanism to establish a one-time key, after which traffic on the interface is secured using an encryption protocol. The configuration interface does not establish a permanent communication channel. Instead, it encodes the one-time key into each transfer and relies on shared secret domain and credential information to provide for key reconstruction at the receiving end.

The fundamental security of both interfaces is dependent upon the chosen data encryption algorithms. In the prototype system, known encryption standards were chosen, which bring with them known levels of security. The Blowfish scheme used for the prototype configuration data encryption is susceptible to attacks on potential weak keys in reduced-round variants and outright deciphering of 3-round variants as described in [70]. The authentication interface of the prototype system employs the RSA algorithm [71] with a 512-bit key for encryption. The security of RSA is dependent upon the difficulty posed in factoring large numbers. A factoring of a 512-bit product of two 78-bit primes was done by a group of researchers led by Herman te Riele of CWI in Amsterdam [72]. The effort took seven months using 300 SGI, Sun, and Pentium PC workstations along with a Cray C916 mainframe. Other potential attacks on the RSA algorithm are summarized in [73].

Brute force attacks on these links are not practical since it would take several centuries to decode a single message using today's computing resources. The fact that no textual information is transferred over these links makes an attacker's job more difficult, as brute force attacks depend upon being able to recognize the fact that the data has been decrypted. In this case, that would amount to recognizing the packet structure by noting the presence of a known domain identifier, author identifier, or credential identifier in the configuration information or a known configuration identifier in the authorization interface data. The ability to recognize this information would imply that security has otherwise been compromised to the extent that some fixed identification numbers are known.

The attack graphs depicted in Figures 6.10 through 6.12 demonstrate the sequence of events that must occur for two different types of attack on this system. Figures 6.10 and 6.11 develop attacks that are aimed at discovering the function embedded in the secure adaptive network processor for a single valid user. Figure 6.12 outlines an attack aimed at providing that user with a malicious function. In all graphs, red nodes represent particular challenges to the attacker. Failure to meet any single one of these challenges will be enough to render the attack ineffective.

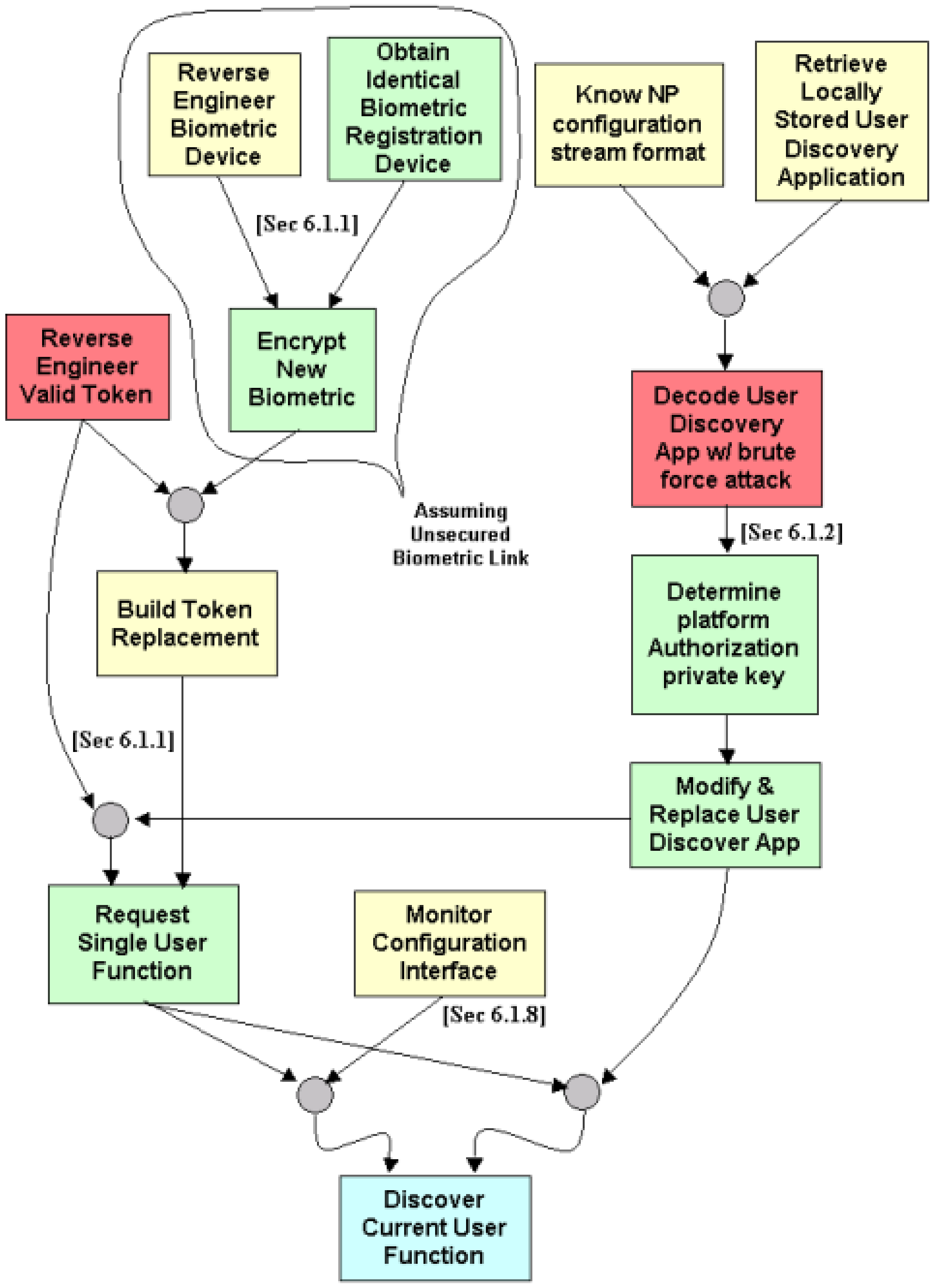

Reverse Engineer Biometric Device
Obtain Identical Biometric Registration Device
Know NP configuration stream format
Retrieve Locally Stored User Discovery Application
[Sec 6.1.1]
Reverse Engineer Valid Token
Encrypt New Biometric
Decode User Discovery App w/ brute force attack
Assuming Unsecured Biometric Link
[Sec 6.1.2]
Build Token Replacement
Determine platform Authorization private key
[Sec 6.1.1]
Modify & Replace User Discover App
Request Single User Function
Monitor Configuration Interface
[Sec 6.1.8]
Discover Current User Function

Figure 6.10: *Single user IP discovery attack*

6.2 Objectives Reexamined and Device Comparison

The objectives of this effort were to demonstrate the viability of a secured run-time reconfigurable processor for streaming-data communication applications and to develop structured methods for assembling and securely reconfiguring this type of processor. The resultant system was to be resistant to all board-level attacks aimed at discovering the

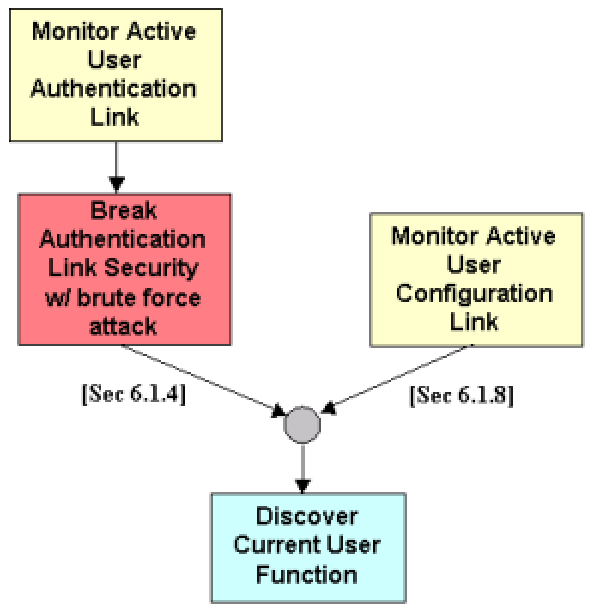

Monitor Active User Authentication Link
Break Authentication Link Security w/ brute force attack
Monitor Active User Configuration Link
[Sec 6.1.4]
[Sec 6.1.8]
Discover Current User Function

Figure 6.11: *Alternate single user IP discovery attack*

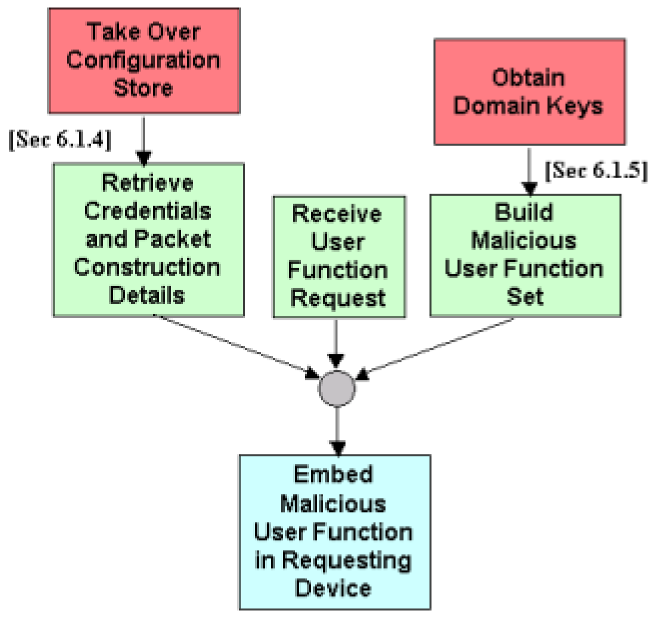

Take Over Configuration Store
[Sec 6.1.4]
Obtain Domain Keys
[Sec 6.1.5]
Retrieve Credentials and Packet Construction Details
Receive User Function Request
Build Malicious User Function Set
Embed Malicious User Function in Requesting Device

Figure 6.12: *Malicious function embedding attack*

internal structure of the device. Toward this end, three design maxims were observed: all external interfaces must be secured; hardware details must be unobservable; the device must contain no secured intellectual property unless a valid user is present. This section examines each of those maxims as they apply to the ideal secure adaptive network processor described in Chapters 3 and 4 and compares the operation of this device with some of the alternative network processors discussed in Chapter 2.

6.2.1 Interfaces must be secured

The secure adaptable network processor structure defined in Chapter 4 has two primary interfaces involved in transferring protected device operational content to the system – the authentication interface and the Configuration Store interface. Both of these connections must be secured to meet the first design maxim. In this case, a trusted relationship is establishing between a user authentication token and the platform, and data is transferred between them in an encrypted format. The Configuration Store interface (and local configuration memory) is protected in a somewhat more flexible and robust manner using one-time keys based upon user credentials. This technique not only protects data from bus snooping attacks, but also reduces the usefulness of power and timing analysis attacks as key management is done in a parallel-tasking hardware-based system and data is transferred in a single block prior to any decoding or key construction operation (Section 3.1). Section 6.1 provided a detailed analysis of potential attacks on these interfaces.

Although contemporary network processors do not specifically include support for hardware security, some systems do have an inherent resistance to attacks aimed at discovering system operation via device interfaces. For example, the IBM PowerNP2G contains internal flash memory for program storage. If a system can be constructed using this device where all operational code is stored in the internal memory, that system will be less vulnerable to bus monitoring attacks than a design that requires an external program stream, although it will be less resistant to offline reverse-engineering attacks. The USC/ISI GRIP system and the Washington University FPX modules are certainly susceptible to bus monitoring attacks during the programming phase, but power and timing analysis attacks are less likely to succeed due to the structural hardware-based approach to the design of both entities. Table 6.2 compares the resistance to various attacks provided by systems built upon the network processing devices discussed in Chapter 2 and the ideal secure adaptable network processor described in Chapters 3 and 4. In this table, devices with external instruction streams are seen to be vulnerable to bus monitoring. Devices built around a known processor base are likewise seen to be susceptible to power and timing analysis attacks. Finally, since they make no effort to verify data sources, all of the network processing devices other than the secure adaptable processor can be easily subjected to any number of substitution attacks including interception and modification of data and/or instruction streams.

Table 6.2: *Attack resistance matrix*

Device	Attack Resistance			
	Bus Monitoring	Power Analysis	Timing Analysis	Substitution
Intel IXP 2850				
IBM PowerNP2G	✓			
Motorola C-3e				
Vitesse IQ2200				
Chameleon CS2112		✓	✓	
Agere APP750NP				
PMC Sierra RM9000 (General-Purpose Processor)				
IDT PAXport 2500				
USC/ISI GRIP		✓	✓	
WU FPX		✓	✓	
Ideal Secure Adaptable NP	✓	✓	✓	✓

6.2.2 Hardware details must be unobservable

Concealing the hardware functionality of a device involves two aspects. Firstly, the configuration or program stream details must be concealed as the device is operating. Secondly, the device must be protected against offline attacks that seek to discover operational details by physically examining the device. Software-based systems are typically susceptible to the first of these attacks, where program details and device operation can be observed by monitoring the operational system and instruction stream. Removal of power from these systems does stop the flow of instruction data into the processor, but does not eliminate the local storage of those instructions or conceal the structural details of the processor used to execute them. In standard hardware-based solutions, the device operation is easily discovered by physically examining the offline device, as it remains even when power is removed. Hardware-based solutions are typically less susceptible to bus monitoring attacks, however, as they do not require an instruction stream and do not present characteristic power or timing signatures for different data content.

The secure adaptable network processor described in this document contains elements that protect it from both online and offline attacks on operational details. The authorization and configuration interfaces are secured during operation to reduce the effectiveness of online attacks in which devices are substituted, modified, or monitored. The analysis of Section 6.1 highlights the strengths and weaknesses of the security schemes used to conceal information flowing on external interfaces that may reveal hardware details.

As a reconfigurable hardware device, the secure adaptable network processor can contain a wide variety of network support in a single device. Data classification, routing, packetization, and processing can all be encapsulated in a single device at once, thus minimizing external data flow connections. This feature helps in system design, as fewer data connections need to be secured between processing stages than might be needed in a system based on a set of fixed-function devices or general-purpose processors. The usefulness of power and timing analysis attacks is also reduced as the user function embedded in a working system is hardware-based and does not need to depend on a known processor core.

6.2.3 Device must contain no IP unless a valid user is present

The adaptable nature of the secure network processor presented here allows the network processor to adapt to user needs as they arise. It also allows the system to remove all traces of that user function when valid users are not present. Offline attacks, such as those used to examine ASIC contents, analyze system content while a device is not operating. The effectiveness of these attacks is reduced in the secure adaptable network processor by removing all user function IP from the system when a user is not present. The user discovery application does remain in the device, however, and could potentially be used to assist in the platform replacement attacks of Sections 6.1.2 and 6.1.6. This application is a good candidate for further study. Weaknesses in the token or Configuration Store protection could also be used to assist in attacks on the network processor. Additional study of these external devices should be done to strengthen the overall systems designed around this processor.

An additional benefit of the removal of user IP from the idle device is that the system becomes inherently adaptable to differing user needs. The system adapts to individual needs as each user activates the system with their desired function. Furthermore, built-in support for secure retrieval of configuration data provides user applications with the means to retrieve additional user function as needed. In other words, systems built around this device provide built-in support for run-time reconfiguration.

6.2.4 Objective Summary and Comparisons

A network processing system designed around the secure adaptable network processor presented in this dissertation provides several advantages over conventional network processor-based systems. Since it was developed with hardware security in mind, it allows for system designs that much more effectively conceal operational details. Figure 6.13 compares the level of security based upon this functional concealment. In this figure, devices that require additional support components are considered less secure, as systems constructed around them require more exposed (and unsecured) interfaces.

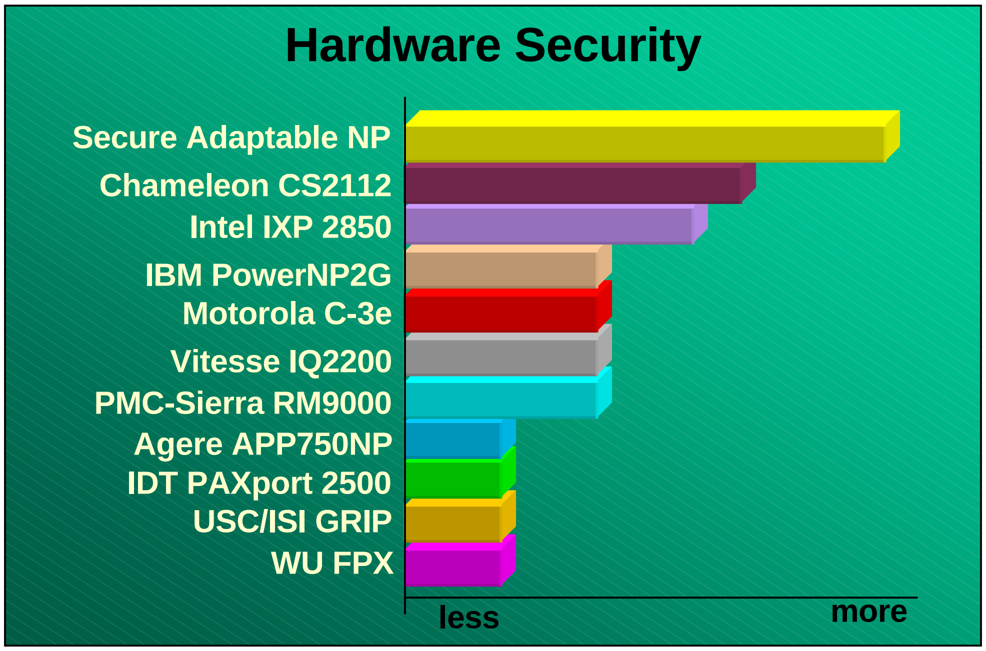


Figure 6.13: *Hardware security comparison*

Fixed-function ASIC devices are inherently insecure, as they not only require external devices for production of a useful network processing solution, but their hardware nature leaves them susceptible to offline reverse engineering. Processor-based systems have the potential for slightly increased security since more functionality might be contained in a single device. Unfortunately, their reliance on external instruction streams means that they are susceptible to instruction bus monitoring attacks while operational. While offline observation of the devices themselves might reveal very little about these processor-based systems, attacks that determine the contents of instruction storage are particularly effective. Devices like the Intel IXP 2850 reduce the effectiveness of operational bus monitoring for applications that can fit in the internal instruction storage space. They do, however, still remain vulnerable to offline analysis of this instruction store. Finally, reconfigurable systems like the USC/ISI GRIP are particularly vulnerable to IP theft, as they provide the hardware functionality in the form of an unsecured configuration stream that could easily be used in other devices to provide the same function. These systems could easily increase security somewhat by using current vendor bitstream encryption, or more fully by adapting the interface structure described in this document, however.

As described in Section 6.2.3, the flexible hardware basis for the secure adaptable network processor permits the construction of systems fully self-contained within a single device. This is a consideration in the security of the system as discussed above. It is also a consideration when attempting to minimize components in a network processing design. Figure 6.14 is a comparison of the current network processor systems based simply on the criteria of the amount of a system that can be contained in a single device. The least self-contained systems, such as the IDT PAXport 2500 classifiers are designed to simply

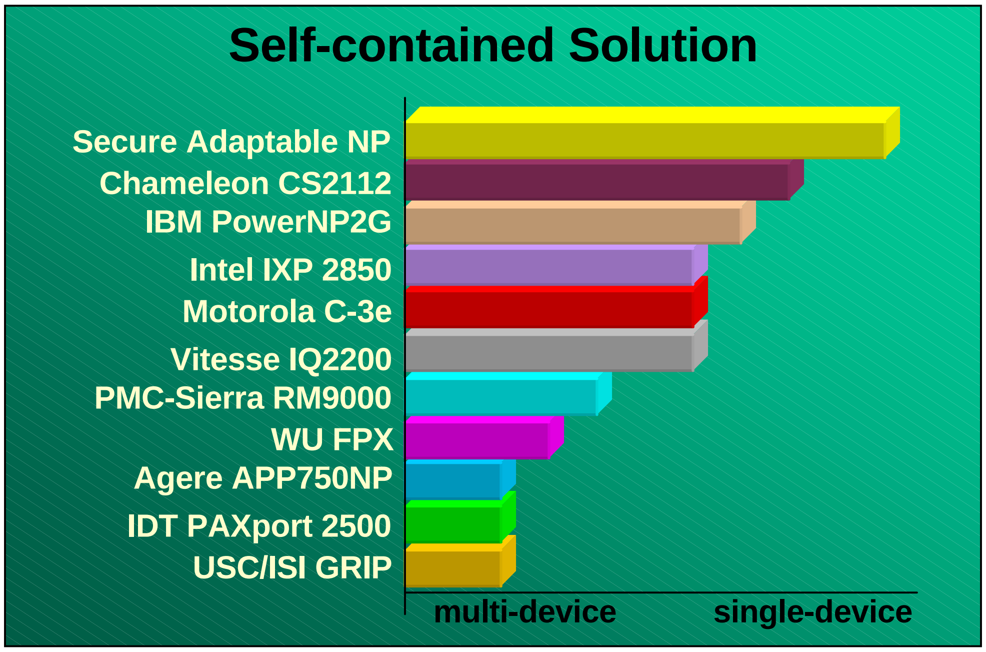


Figure 6.14: *System self-containment by device*

provide a portion of a larger solution. Devices based upon processing cores provide more self-contained solutions, as a wider variety of system function can be performed in a single component. Many of these devices do not, however, contain basic network interfaces and still require off-chip storage of instruction streams during system operation. The on-chip storage and MAC support of the IBM PowerNP2G allows for a somewhat more self-contained solution. Reconfigurable systems provide the most potential for self-contained systems. The Chameleon CS2112 does not require external instruction data, and provides a reconfigurable processing base that can implement a variety of network functions. In a system requiring user authentication, it may fall short, however, as no internal security functions are included. Finally, reconfigurable systems like the USC/SI GRIP are designed using several components, and are inherently multi-device solutions. It would certainly be possible, however, to incorporate the function that they provide into a single system like the secure adaptable network processor to make them a truly single-device solution.

Lastly, the dynamic nature of this system allows for an increased level of dynamic user adaptability. Figure 6.15 compares the adaptability of this system to that of other devices. Clearly, reconfigurable solutions like the USC/ISI GRIP could be as adaptable as the secure network processor. Their current lack of user identification and authorization means that in their current incarnations they are not inherently suited to user adaptation. Some additional support hardware and minor internal modification could remedy this, however. General-purpose processing solutions are likewise not currently user-adaptable. They to could benefit from additional support in this area, but remain somewhat limited in their potential by the restriction to a fixed instruction set. Finally, ASIC devices designed for a specific purpose like the Agere APP750NP may provide some flexibility within their intended function, are clearly not adaptable to provide other functions.

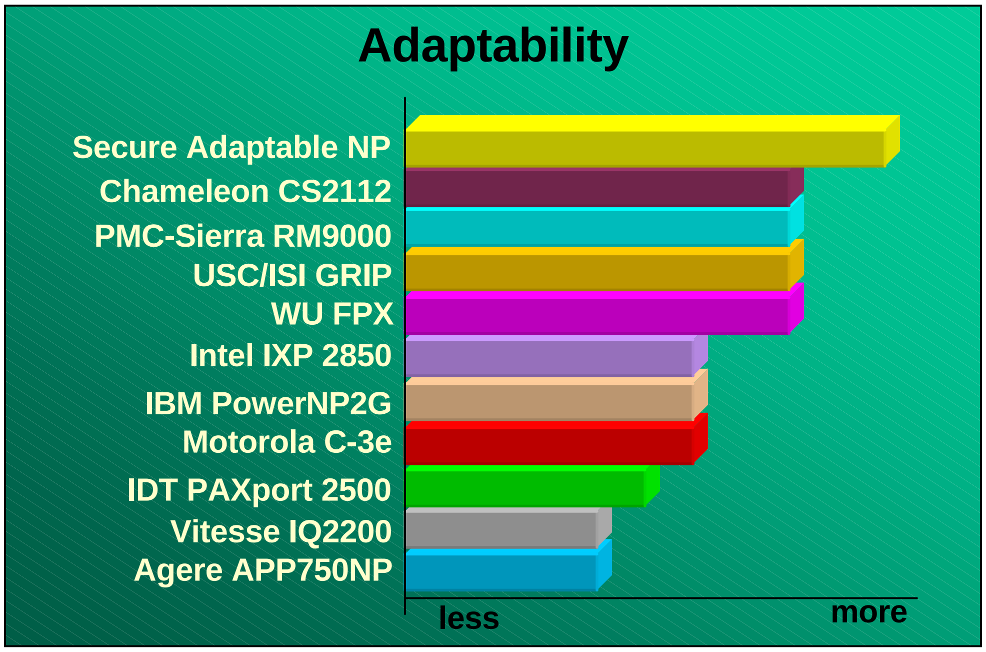


Figure 6.15: *Device adaptability comparison*

As reconfigurable hardware technology continues to improve, the size of standard reconfigurable devices will continue to grow. The space available within current reconfigurable systems allows for custom hardware and standard processor cores to be mixed together on a single device. Newer technology will allow even more functionality within a device, making single-chip solutions even more practical. As this growth trend continues, however, the security structure described in this document remains fixed in size. Since the security support does not require scaling, the ratio of user resources to system resources will continue to improve. This trend toward more complex user systems with fixed security overhead makes this an increasingly attractive feature.

6.3 Contributions

This document describes research related to the design of a secure adaptable network processor. Several objectives were laid out in Table 3.1 at the start of this endeavor, and discussed in terms of the final result at the conclusion (Section 6.2). Along the way, a secure device structure was defined and investigated. Both the utility and security of the device were examined in Chapters 5 and 6. In the course of this research effort, several contributions were made including the following four items.

1. A secure system structure capable of supporting reconfigurable network processing was developed as described in Chapter 4 and implemented using currently available hardware in Chapter 5.

2. A key management technique for securing network processor assembly data and its transfer to the system is provided as outlined in Section 3.2.3.

3. A built-in user-customization and upgrade path for the underlying hardware of a network processing system is provided by the defined system.

4. Two user functions with different characteristics were implemented on a prototype foundation to demonstrate and evaluate the secure network processing concepts introduced in this work.

The secure adaptable network processor described in Chapters 3 and 4 and analyzed in Chapters 5 and 6 provides the basis for these contributions. The following points define this secure device and mechanisms modeled on it.

1. The device contains only two configuration-related interfaces – the authorization interface and the encrypted configuration interface.

    a. Initial bootstrap should be done through the encrypted configuration interface. The use of an alternate interface could provide an unsecured point for system entry.

    b. No other internal (e.g. imap) or external (e.g. selectmap) configuration interface should be made available to the user. Such an interface would allow a user to circumvent the system security.

2. The only way to program/configure the device is through the encrypted configuration interface. The use of any other interface would violate the first point, while the use of the configuration interface in a manner other than the standard encryption technique could provide an unintended access point for attack.

3. No device readback function is provided. The objective is to protect the device content. Allowing it to be read back out is contrary to this notion.

In addition to these points, a device should not accept any application other than the original user discovery application when no user is present. The internal key used to decode the discovery application will provide some assurance that it is the desired configuration. While not addressed here, additional use of a known hash or CRC over the configuration would further assure that the correct information is loaded.

Finally, the applications that are run on a system built around this processor should be verified prior to serving them out from the Configuration Store. An application that causes the device to physically fail due to internal contention or some other poor design practice will still cause this device to fail. The secure adaptable network processor does not protect the system from poor design practices.

6.4 Future Directions

The results of this work demonstrate that a secure network processor is a viable alternative to traditional network processing solutions. The two prototypes included here were valuable in both developing the concepts and evaluating their usefulness. Additional research into applications for this technology would, however, also be beneficial. The unique qualifications of the secure adaptable network processor to run-time adaptation is certainly worth additional exploration.

6.4.1 Full Implementation of Security Scheme

The initial prototype application described in Section 5.2 did not include the full configuration stream security system outlined in Section 3.2.3. The second application discussed in Section 5.3 will include most of the security, but is still under development. As this development proceeds, it may become apparent that some details of the security scheme need to be modified to suit current hardware or to strengthen security. In the first case, the changes should be removed in following prototypes, in the second, updates should be noted and documented elsewhere.

6.4.2 Configuration Store and Tokens

Additional techniques for local data protection on the Configuration Store, including self-erasure if intrusion is detected and methods of securing the initial transfer of encrypted configuration data to the Configuration Store are worthy of further consideration. A method for distribution of domain configuration keys to IP creators needs to be considered as well, since loss of these keys would reduce the security of data held on the Configuration Store.

Techniques for securely moving data into token devices and protecting the data once it gets there, including encryption and self-erasure prior to examination should also be investigated. As indicated in the analysis of Section 6.1.5, the integrity and physical security of the both the tokens and the token creation system is an important factor in maintaining overall system security. In a related issue, the issue of key management within the system needs to be addressed as a significant amount of credential and key information is manipulated as users are added and removed and credentials are created and expire.

6.4.3 Single-device Packaging

The prototype system described in Chapter 5 spans three FPGA devices. While the separation of functionality does provide an excellent base for modular system development and testing, it does not fully express the single-device secure adaptable processor described in Chapters 3 and 4. The functionality of the three devices should be folded in to a single device in the next generation prototype to more fully demonstrate the described processor.

At the outset of this research effort, the Xilinx *Virtex* FPGA and the ISI/EAST SLAAC1V prototyping platform represented the leading edge of commercial reconfigurable technology available. During the course of the research effort, other large reconfigurable devices have become available, including the Xilinx *Virtex-II*. As mentioned in Section 4.1, the *Virtex-II* family of reconfigurable devices provides internal access to the configuration ports. This access allows for self-reconfigurable systems, and will permit the functionality of all three current prototype FPGAs to be folded into a single device. As work continues on the secure adaptable network processor, techniques for self-reconfiguration of *Virtex-II* devices are being examined ([74]), and it is expected that the next prototype will fold the function of the three *Virtex* FPGA parts into a single *Virtex-II* part.

6.4.4 Cores for General Implementation

To promote the manufacture of a secure adaptable network processor device, the existing system components should be packaged as portable cores and additional data processing core elements should be developed. Much of this component packaging has already been done due to the modular method of system development undertaken for the prototype systems. The ability to easily transport these cores and embed them at run-time is still under investigation, however, as the system is retargeted toward a single device package. New cores for user processing should be developed to provide the foundation of additional user applications.

6.4.5 Applications

Finally, with the second prototype nearing completion, additional applications for this platform should be considered as a third-generation prototype begins construction. Demonstrations of data processing on the device, including in-packet data encryption and compression would be excellent candidates for highlighting features of the system. A configurable router that compresses data onto a lower-bandwidth wireless link might be an attractive candidate.

Applications that make use of multi-channel differentiated services within the processor would serve to highlight both the reconfigurable processing channels and user-based configuration. A system that makes use of one channel as a path for reconfiguration data would be an interesting example of this type of application. This application could provide runtime updates to the user application via wired or wireless connections. Some support for this truly dynamic type of system has been included in the current prototype structure (Figure 5.8), as one of the reconfigurable processing channels is connected to the authentication engine.

BIBLIOGRAPHY

[1] Thompson, J.F., J. Bernstein, and J. Crane, "*Identity Theft*," Presentation to the President's Information Technology Advisory Committee (PITAC) 12th meeting, February 7, 2001.

[2] Evers, J., "Dutch police fight cell phone theft with SMS bombs," IDG News Service \ Amsterdam Bureau, March 27, 2001.

[3] Smart, N.P., "Physical Side-Channel Attacks on Cryptographic Systems," *Software Focus*, volume 1, issue 2, December 2000, pp. 6-13.

[4] Kocher, P., J. Jaffe, and B. Jun, "Differential Power Analysis," *Lecture Notes in Computer Science: Advances in Cryptology – Crypto '99,* Springer LNCS, volume 1666, 1999, pp. 388-397.

[5] Kocher, P.C., "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," *Lecture Notes in Computer Science: Advances in Cryptology – Crypto '96*, Springer LNCS, volume 1109, 1996, pp. 104-113.

[6] Juniper Networks, *T-Series Routing Platform Data Sheet*, Part Number 100051-0006, Juniper Networks Inc., Sunnyvale, CA, February 2003.

[7] Austrian Aerospace, "*Spacewire Router ASIC Development*," ESM-006, Presentation to the European Space Agency (ESA) Spacewire Working Group, September 18, 2001.

[8] May, M.D., P.W. Thompson, and P.H. Welch, ed., *Networks, Routers and Transputers: Function, Performance and Applications*, IOS Press, Headington Burke, VA 1993.

[9] Smitt, E. L. and R.J. Collins, "Microprocessor based control and switching device," United States Patent no. 4,685,124, assigned to Data General Corporation, Westboro, MA, April 30, 1985.

[10] Lawson, S., “Network processors enter new generation,” IDG News Service, *Network World Fusion,* June 19, 2002

[11] Husak, D., “Programmable NPUs ‘edge’ out ASICs,” *EE Times*, November 18, 2002.

[12] Foremski, T., “Network chips: battleground for the big boys,” *Financial Times Survey*, FT.com, April 15, 2002.

[13] Blythe, S., B. Fraboni, S. Lall, H. Ahmed, and U. de Riu, “Layout reconstruction of complex silicon chips,” *IEEE Journal of Solid-State Circuits*, volume 28, pp. 138 – 145, February 1993.

[14] Huang, A., “Keeping Secrets in Hardware, the Microsoft XBox^{TM} Case Study,” *CHES 2002*, August 2002.

[15] Erickson, C.R., D. Tavana, and V. A. Holen, “Encryption of Configuration Stream,” United States Patent no. 6,212,639 B1, assigned to Xilinx Inc., San Jose, CA, April 3, 2001.

[16] Batinic, I., L. Kraus and M. P. Loranger, “Field Programmable Gate Array With Program Encryption,” United States Patent no. 6,351,814 B1, assigned to Credence Systems Corporation, Fremont, CA, February 26, 2002.

[17] Anderson, R. and M. Kuhn, “Tamper Resistance – a Cautionary Note,” Proceedings of *The Second USENIX Workshop on Electronic Commerce*, Oakland, CA, November 18-21, 1996, pp 1-11.

[18] Kanellos, M., “Intel strides into network chips with new products, fund, “ CNET News.com, September 1, 1999.

[19] Intel Corp., *Intel IXP1200 Network Processor Data Sheet*, Intel Corporation, Part number 278298-010, December 2001.

[20] Intel Corp., *Intel IXP1240 Network Processor Data Sheet*, Intel Corporation, Part number 278405-003, December 2001.

[21] Intel Corp., *Intel IXP1250 Network Processor Data Sheet*, Intel Corporation, Part number 278371-006, December 2001.

[22] Feghali, W., B. Burres, G. Wolrich, and D. Carrigan, “Security: Adding Protection to the Network via the Network Processor,” *Intel Technology Journal*, volume 6, number 3, August 15, 2002, pp. 40-49.

[23] IBM Corp., *IBM PowerNP NP4GS3 Network Processor Data Sheet (Preliminary)*, International Business Machines Corp. Microelectronics Division, Hopewell Junction, NY, January 29, 2003.

[24] Chandra, V., "Selecting a network processor architecture," IBM Microelectronics Technology Group, August 2002.

[25] IBM Corp., *IBM PowerNP NP2G Network Processor Data Sheet (Preliminary)*, IBM Microelectronics Division, Hopewell Junction, NY, February 12, 2003.

[26] C-port Corp., *C-5 Network Processor Architecture Guide*, C-port technical library C5NPD0-AG/D, C-port Corporation, North Andover, MA, May 2001.

[27] Motorola, *C-3e Network Processor, Silicon Rev A1, Data Sheet*, C3ENPA1-DS/D Rev 3, Preliminary, Motorola Inc., 120 Water Street, No. Andover, MA, November 2002.

[28] XaQti Corp., *XQ1200BG GigaPOWER Protocol Processor Product Preview*, document 1200-0498-02, XaQti Corporation, San Jose, CA., 1998.

[29] Vitesse Semiconductor, *IQ2200 Product Brief*, Vittesse Semiconductor Corp, Camarillo, CA, 2002.

[30] Chameleon Systems, *Wireless Base Station Design Using Reconfigurable Communications Processors*, Chameleon Systems Inc., San Jose, CA, 2000.

[31] Wirbel, L., "Network processors take reconfigurable approach," *EE Times*, May 22, 2000.

[32] Agere Systems, *Smart Processing for Terabit Networks, PayloadPlus Processor Family Overview*, Agere Systems Incorporated, Allentown, PA, 1999.

[33] Agere Systems, *10G Network Processor Chip Set (APP750NP and APP750TM) Product Brief*, Agere Systems Incorporated, Allentown, PA, November 2002.

[34] PMC-Sierra, *RM9000x2 Integrated Multiprocessor Data Sheet (preliminary)*, PMC-Sierra Incorporated, Burnaby, B.C., 2001.

[35] Solidum Systems, *PAX.port 2500 Product Brochure*, IDT Canada Inc., Ottawa, Ontario, 2002.

[36] Bellows, P., V. Bhaskaran, J. Flidr, T. Lehman, B. Schott, K. Underwood, "GRIP: A Reconfigurable Architecture for Host-Based Gigabit-Rate Packet Processing," *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM) '02*, Napa, CA, April 2002, pp. 121-130.

[37] Xilinx, Inc., *Virtex Data Sheet*, Xilinx Inc., San Jose, CA, February 2000.

[38] Lockwood, J.W., “An Open Platform for Development of Network Processing Modules in Reprogrammable Hardware,” *IEC DesignCon ‘01*, Santa Clara, CA, January 2001, paper WB-19.

[39] Sproull, T., J. Lockwood, and D. E. Taylor,, “Control and Configuration Software for a Reconfigurable Networking Hardware Platform,” *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM) ‘02*, Napa, CA, April 2002, pp. 45-54.

[40] Memik, G., S. O. Memik, and W. H. Mangione-Smith, “Design and Analysis of a layer Seven Network Processor Accelerator Using Reconfigurable Logic,” *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM) ‘02*, Napa, CA, April 2002, pp. 131-140.

[41] Brebner, G., “Single-chip Gigabit Mixed-version IP Router on Virtex-II Pro,” *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM) ‘02*, Napa, CA, April 2002, pp. 35-44.

[42] Xilinx Inc., *Virtex-II Pro Platform FPGAs: Introduction and Overview*, DS083-1 v2.4.1, Advance product specification, Xilinx Incorporated, San Jose, CA, March 24, 2003.

[43] EE6504: Rapid Prototyping of Computing Machinery – The Hokie RISC, http://www.ee.vt.edu/courses/ee6504/rapid.html, Bradley Dept of Electrical Engineering class, Virginia Polytechnic Institute and State University, December 1995.

[44] Swanchara, S., S. Harper, and P. Athanas, "A Stream-Based Configurable Computing Radio Testbed," *IEEE Symposium on FPGAs for Custom Configurable Computing Machines*, Napa, California, April 1998, pp. 40-47.

[45] Lee, D.C., S. Harper, P. Athanas, and S.F. Midkiff, "A Stream-based Reconfigurable Router Prototype," *IEEE International Conference on Communications*, Vancouver, B.C., June 1999, pp. 581-585.

[46] Hansen, S. “Rugged, Biometric “Match-On-Device” Token In R&D at Datakey Electronics, Inc.,” http://www.datakeyelectronics.com/PR6.htm, Datakey Electronics, Inc., Burnsville, MN, August 14, 2002, last accessed April 3, 2003.

[47] Abraham, A., *It is I, An Authentication System for a Reconfigurable Radio*, M.S. thesis, Virginia Polytechnic Institute and State University, August 2, 2002.

[48] TecSec Inc., *CKM Technology Overview*, white paper, TecSec Incorporated, Vienna, VA, April 2001.

[49] Dallas Semiconductor, *Preliminary DS-1957B Java Powered iButton Data Sheet*, Dallas Semiconductor Corporation, July 22, 2002.

[50] Xilinx Inc., *Virtex Series Configuration Architecture User Guide*, Xilinx, Inc, San Jose, CA, February, 2000.

[51] Xilinx Inc., *Virtex-II Handbook*, Xilinx, Inc., San Jose, CA, November 2002.

[52] Sundararajan, P., and S. A. Guccione, "XVPI: A Portable Hardware / Software Interface for Virtex," *Reconfigurable Technology: FPGAs for Computing and Applications II, Proc. SPIE 4212*, Bellingham, WA, November 2000, pp. 90-95.

[53] ISI/East, *SLAAC1-V VHDL Users Guide, Release 0.1.3*, University of Southern California Information Sciences Institute/East, 2001.

[54] SecuGen Corporation, *SecuGen FDA01 Developer's Guide, DC1-0001B Rev A*, SecuGen Corporation, Milpitas, CA, April 24, 2001.

[55] Schneier, B., "*Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)*," Fast Software Encryption, Cambridge Security Workshop Proceedings (December 1993), Springer-Verlag, 1994, pp. 191-204.

[56] Landaker, W.J., "Free Blowfish VHDL Code," SourceForge, http://sourceforge.net/projects/blowfishvhdl/, registered October 18, 2000, development status 4-Beta, last accessed April 3, 2003.

[57] Counterpane Labs, "Counterpane Labs: Blowfish," http://www.counterpane.com/blowfish.html, last accessed April 3, 2003.

[58] Wesson, R., F. Hayes-Roth, J. W. Burge, C. Stasz, and C. A. Sunshine, "Network structures for distributed situation assessment," *IEEE Trans. Systems, Man, and Cybernetics*, volume SMC-11, no. 1, January 1981, pp. 5-23.

[59] Iyengar, S.S., D. N. Jayasimha, and D. Nadig, "A versatile architecture for the distributed sensor integration problem," *IEEE Transactions on. Computing*, volume 43, number 2, February 1994, pp. 175-185.

[60] Jones, M.T., S. Mehrotra, and J. H. Park, "Tasking Distributed Sensor Networks," *International Journal of High Performance Computing Applications*, volume 16, number 3, August 2002, pp. 243-258.

[61] Akyildiz, I.F., Weilian Yu, et. al. "A Survey on Sensor Networks," *IEEE Communications Magazine*, August 2002, pp. 102-114.

[62] Carman, D.W., P. S. Kruus, and B. J. Matt, "*Constraints and Approaches for Distributed Sensor Network Security*," NAI Lab Technical Report 00-010, September 2000.

[63] Digital Wireless Corporation, *WIT2410 2.4GHz Spread Spectrum Wireless Industrial Transceiver Integration Guide*, June 14, 2000, Digital Wireless Corporation, Norcross, GA.

[64] Z-World Inc., *RabbitCore RCM2100 Series: User's Manual, part number 019-0091, rev. 010315A-A*, Z-World Inc, Davis, CA, 2001.

[65] RFC 760, "DOD Standard Internet Protocol," prepared by Information Sciences Institute, University of Southern California, January 1980, p. 14.

[66] Red River Engineering, *WaveRunner PMC Model 301 Datasheet*, Red River Engineering, Richardson, TX.

[67] Tanenbaum, A. S., *Computer Networks, 2nd Edition*, Prentice Hall, Englewood Cliffs, NJ, 1988, p. 212.

[68] Wiznet Inc., *IIM7010 Product Description*, Wiznet Incorporated, Kangnam-ku, Seoul, Korea.

[69] Wiznet Inc., *i2Chip W3100A Technical Datasheet v1.3*, Wiznet, Incorporated, Kangnam-ku, Seoul, Korea.

[70] Schneier, B., "The Blowfish Encryption Algorithm – One Year Later," Dr. Dobb's Journal, volume 20, number 9, September 1995, p. 137-138.

[71] Rivest, R.L., A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[72] Schneier, B., "Factoring a 512-bit number," *Crypto-gram Newsletter*, Counterpane Internet Security Inc., Cupertino, CA, September 15, 1999.

[73] Kaliski, B.S. Jr. and M.J.B. Robshaw, "The secure use of RSA," *CryptoBytes*, volume 1, number 3, RSA Laboratories, Redwood City, CA, Autumn 1995, pp. 7-13.

[74] Fong, R.J., S.J. Harper, and P.M. Athanas, "A Versatile Framework for FPGA Field Updates: An Application of Partial Self-Reconfiguration," *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping*, San Diego, CA, June 2003.

GLOSSARY

AES: *Advanced Encryption Standard* (previously known as the Rijndael Algorithm) that supports variable block lengths and key sizes.

ARC: 32-bit RISC/DSP processor core from ARC International, Elstree, UK.

ASIC: *Application-Specific Integrated Circuit*. A digital semiconductor device designed to provide a specific function.

Bitstream: A set of data used to configure an FPGA for operation.

Blowfish: 64-bit block cipher encryption algorithm providing key lengths from 32 bits to 448 bits.

CRC: *Cyclic Redundancy Check*. An error detection scheme used for digital data transmission.

Credential: Access permissions assigned to and/or identifying individual users.

DES: *Data Encryption Standard* (previously known as the Lucifer algorithm). A 1976 standard encryption algorithm specified by the US National Institute of Standards and Technology designed for 64-bit blocks using a 64-bit key.

DoS: *Denial of Service* attack. An attack in which a device is kept busy with non-essential operations so that normal operation slows or halts.

DSN: *Distributed Sensor Network*. A connected set of sensor devices that provides for intelligent processing and forwarding of sensor data.

ECC: *Error Correcting Code*. An error correction scheme used for digital data transmission.

FIFO: *First In First Out* storage (queuing).

FPGA: *Field Programmable Gate Array*. A digital semiconductor device containing a set of elements that can be (re)programmed in the field.

ICAP: *Internal Configuration Access Port*. An internal access point for the SelectMAP interface in a Xilinx *VirtexII* part.

IP: *Intellectual Property*. Ideas with commercial value including patented and copyrighted material.

MAC: *Medium Access Control*. The fundamental (underlying) data transfer protocol for a network system (e.g. Ethernet).

MIPS: MIPS Technologies Incorporated (Mountain View, CA) RISC processor core.

NAVCIITI: *Navy Collaborative Integrated Information Technology Initiative*. A project initiative sponsored by the Office of Naval Research intended to accelerate certain research activities involving communications and information technologies.

Network Processor: a reprogrammable processing device that is designed specifically for use in a networking environment

ONR: United States Office of Naval Research.

PC: *Personal Computer*. A computer designed for use by an individual.

PDA: *Personal Digital Assistant*. A portable device used for personal organization.

PHY: *Physical Layer*. The protocol used for physical connection to, and raw bit transfer over, a network medium.

RISC: *Reduced Instruction Set Computer*. A computer processor design that uses relatively few instructions.

Run-time: The time period when a system is actively engaged in processing.

SelectMAP: *Selectable Microprocessor Access Port*. An 8-bit bidirectional configuration port available on Xilinx configurable logic devices.

SHA: *Secure Hash Algorithm*. A condensed representation of a data set that is considered secure due to the computational infeasibility of reproducing the data from the representation.

SLAAC: *Systems Level Applications of Advanced Computing*. The acronym associated with a reconfigurable platform produced by the University of Southern California Information Sciences Institute – East.

SRAM: *Static Random Access Memory*. A memory device that does not retains information while power is applied.

Token: An identification device carried by an individual. Identifies both the person and their desired system function.

XVPI: *Xilinx Virtex Portable Interface*. A hardware/software method for using the Xilinx *Virtex* SelectMap interface for device configuration.

APPENDICES

APPENDIX A

SLAAC1-V XVPI PROGRAMMER

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

-------------------------------------------------------------------------
--  Program an FPGA on the SLAAC1V via XVPI interface from X0
--  Requires modification of ISI PCI_Bridge code
--  Scott Harper, Virginia Tech Dept of ECE, July 10, 2002
-------------------------------------------------------------------------


entity xvpi_prog is

  port (
    CLK         : in    std_logic;
    RESET       : in    std_logic;

    TRANSFER    : in std_logic;
    HOLD      : in std_logic;
    PARTIAL    : in std_logic;
    DEVICE_SEL  : in std_logic_vector(2 downto 0);
    TRANS_DATA  : in std_logic_vector(31 downto 0);
    NEW_DATA    : out std_logic;

    XVPI_ADDR    : out std_logic_vector(7 downto 0);
    XVPI_DATA_WR  : out std_logic_vector(20 downto 0);
    XVPI_DATA_RD  : in std_logic_vector(20 downto 0);
    XVPI_WR      : out std_logic;
    XVPI_STB      : out std_logic;
    XVPI_SRC      : out std_logic;

    STATUS         : out   std_logic_vector(1 downto 0)
    );
end xvpi_prog;


architecture STATE_MACHINE of xvpi_prog is

  -- Control Signals
  signal flush_outpipe :  std_logic_vector(1 downto 0);
--  retries  : std_logic_vector(1 downto 0);
  signal trans_byte : std_logic_vector(1 downto 0);
  signal activate, running, waiting : std_logic;

  -- Clocks
  signal stb_clk  : std_logic;  -- half of input clock
  signal config_clk : std_logic; -- half stb_clk when writing config data

  -- Transfer Signals
  signal device_addr  : std_logic_vector(7 downto 0);
  signal   check_byte, check_byte_d,
        check_byte_dd, check_byte_ddd : std_logic_vector(7 downto 0);
  signal trans_data_reg : std_logic_vector(31 downto 0);
  signal xvpi_wr_int : std_logic;
  signal xvpi_addr_int : std_logic_vector(7 downto 0);
  signal write_done : std_logic;

  -- XVPI input bitfields
  signal xvpi_top : std_logic_vector(2 downto 0);
  signal xvpi_done, xvpi_busy, xvpi_cclk : std_logic;
  signal xvpi_mode : std_logic_vector(2 downto 0);
  signal xvpi_init_n, xvpi_prog_n, xvpi_cs_n : std_logic;
  signal xvpi_data : std_logic_vector(7 downto 0);
  signal xvpi_wr_n : std_logic;

  type  Program_State IS (
    IDLE,            -- Wait for bit transfer signal
    READ_STATE,        -- Initialize xvpi input register
    CLEAR_DEVICE,      -- wait for init_n=0, read xvpi[1,2,3] status
    WAIT_FOR_INIT_HIGH,  -- read xvpi until init_n = '1'
    START_CLK,        -- start auto_cclk, set prog_n = 0
    DELAY,          -- needed???
    CONFIG_WRITE,      -- write configuration
    STATUS_CHECK,      -- read xvpi, check for done = '1'
    FAILED,          -- any error goes here
    SUCCESS);        -- end configuration
   signal transfer_state : Program_State;


  -- From xvpi package
  constant  ADDR_X1VPI_REG : std_logic_vector(7 downto 0) := "00000101";
  constant  ADDR_X2VPI_REG : std_logic_vector(7 downto 0) := "00000110";
  constant  ADDR_AUTO_CCLK : std_logic_vector(7 downto 0) := "00010100";
  constant  ADDR_LED_REG   : std_logic_vector(7 downto 0) := "00110000";

--------------------------------------------------------------------------
--------------------------------------------------------------------------
--------------------------------------------------------------------------

begin
--------------------------------------------------------------------------
-- Clock Division
--------------------------------------------------------------------------
  Clock_Divider: process(CLK, RESET)
    begin
    if (RESET = '1') then
      stb_clk <= '0';
    elsif (CLK'event and CLK = '1') then
      stb_clk <= not stb_clk;
    end if;
    end process;
  Clock_Divider2: process(RESET, stb_clk, transfer_state, hold)
    begin
    if (RESET = '1') then
      config_clk <= '0';
    elsif (stb_clk'event and stb_clk = '1') then
       if (hold = '0' and
         (transfer_state = CONFIG_WRITE)) then
      config_clk <= not config_clk;
      end if;
    end if;
    end process;

-------------------------------------------------------------------------
-- Constant/Muxed XVPI signals
-------------------------------------------------------------------------
  activate <= transfer or running;
  device_addr <=  ADDR_X1VPI_REG when DEVICE_SEL = "010" else
            ADDR_X2VPI_REG when DEVICE_SEL = "100" else
            (others => '1'); -- unused address

  xvpi_cclk <= config_clk;
  xvpi_wr_n <= not xvpi_wr_int when transfer_state = CONFIG_WRITE else
'1';
  XVPI_ADDR <=  xvpi_addr_int;
  XVPI_DATA_WR <=   xvpi_top & "ZZ" & xvpi_wr_n & xvpi_cclk & xvpi_mode &
'Z'
                  & xvpi_prog_n & xvpi_cs_n & xvpi_data;
  XVPI_SRC <= activate;

  new_data <= '1' when trans_byte = "10" and transfer_state =
CONFIG_WRITE and
                config_clk = '0' and stb_clk = '0'
          else '0';


-------------------------------------------------------------------------
-- Registered XVPI signals
-------------------------------------------------------------------------

  Output_Signal_Reg: process(CLK, RESET)
    begin
    if RESET = '1' then

      XVPI_WR <= '0';
      XVPI_STB <= '0';

    elsif (CLK'event and CLK = '1') then

      XVPI_WR <=  xvpi_wr_int;
      XVPI_STB <=  activate and xvpi_wr_int and not stb_clk and not hold;

    end if;
    end process;

  Input_Signal_Reg: process(RESET, CLK, xvpi_wr_int, xvpi_addr_int,
                            activate, device_addr)
    begin
    if (RESET = '1') then
      xvpi_init_n <= '1';
      xvpi_done <= '1';
      xvpi_top <= (others => '0');
      xvpi_mode <= "010";

    elsif (CLK'event and CLK = '0') then
       if (activate = '1' and
          xvpi_wr_int = '0' and xvpi_addr_int = device_addr) then
      -- Read selectmap data in
        xvpi_mode <= "010"; -- XVPI_DATA_RD(13 downto 11);
        xvpi_top <= XVPI_DATA_RD(20 downto 18);
        XVPI_Init_n <= XVPI_DATA_RD(10);
        XVPI_Done <= XVPI_DATA_RD(17);
        end if;
    end if;
    end process;


--------------------------------------------------------------------------
-- State Machine
--------------------------------------------------------------------------
  Transfer_States: process(stb_clk, RESET, transfer, activate, hold)
    begin
    if (RESET = '1') then
      status <= "00";
      running <= '0';

      xvpi_addr_int <= (others => '1');
      xvpi_data <= (others => '0');
      xvpi_wr_int <= '0';
      xvpi_prog_n <= '1';
      xvpi_cs_n <= '1';

      flush_outpipe <= "00";
--      retries <= "00";

      write_done <= '0';
      trans_byte <= "11";
      trans_data_reg <= (others => '0');
      check_byte <= (others => '0'); check_byte_d <= (others => '0');
      check_byte_dd <= (others => '0'); check_byte_ddd <= (others =>
'0');
      transfer_state <= IDLE;

    elsif (stb_clk'event and stb_clk = '1') then

      case transfer_state is

        when IDLE =>

        status <= "01"; -- Transfer beginning
            -- Initialize pointers / counters / indicators
            trans_byte <= "11";
            write_done <= '0';
--            retries <= "00";
            running <= '0';
            -- Set up xvpi register and first state
            xvpi_cs_n <= '1';
            xvpi_addr_int <= device_addr;

            flush_outpipe <= "00";
            xvpi_wr_int <= '0';
            xvpi_prog_n <= '1';
            if (activate = '1') then
              if (PARTIAL = '1') then
                transfer_state <= START_CLK;
              else
                transfer_state <= READ_STATE;
              end if;
            end if;

--            xvpi_wr_int <= '1';
--            xvpi_prog_n <= '0';
--            transfer_state <= CLEAR_DEVICE;


        when READ_STATE =>
        running <= '1';
        -- Strobe Program Bit to reset FPGA
          if flush_outpipe = "10" then
            flush_outpipe <= "00";

            -- Strobe Prog_n (no CS)
            xvpi_prog_n <= '0';
            xvpi_wr_int <= '1';
            transfer_state <= CLEAR_DEVICE;

          else
            -- wait for valid xvpi_data_rd
            flush_outpipe <= flush_outpipe + 1;
          end if;

        when CLEAR_DEVICE =>
        -- Wait for FPGA to clear, then set Program to 1

          if flush_outpipe = "10" then  -- wait for init_n = 0
            if (xvpi_init_n = '0') then
              xvpi_prog_n <= '1';
              xvpi_wr_int <= '1';
              flush_outpipe <= "00";
              transfer_state <= WAIT_FOR_INIT_HIGH;
            end if;
          else  -- wait for valid xvpi_data_rd
            xvpi_wr_int <= '0';
            flush_outpipe <= flush_outpipe + 1;
          end if;

        when WAIT_FOR_INIT_HIGH =>
        -- Wait for FPGA to finish program/clear operation

          if flush_outpipe = "10" then  -- wait for init_n = '1'
            if (xvpi_init_n = '1') then -- init_n = 1
              if (xvpi_done = '1') then -- done = 1
                transfer_state <= FAILED;
              else
                flush_outpipe <= "00";
                transfer_state <= START_CLK;
              end if;
            end if;
          else  -- wait for valid xvpi_data_rd
            xvpi_wr_int <= '0';
            flush_outpipe <= flush_outpipe + 1;
          end if;


        when START_CLK => -- using xvpi(14) = 10101... instead
          -- Stop Auto CCLK
          --  xvpi_data <=   "00000010"; -- enable
          xvpi_data <=   (others => '0'); -- disable
          xvpi_addr_int <= ADDR_AUTO_CCLK;
          xvpi_wr_int <= '1';
          transfer_state <= DELAY;

        when DELAY =>
        status <= "10"; -- Non-deterministic init done,
                        -- beginning programming
          -- Turn xvpi_wr off prior to config write
          xvpi_wr_int <= '0';
          transfer_state <= CONFIG_WRITE;

        when CONFIG_WRITE =>
          if (hold = '0') then
            -- Control xvpi address and data, check for done
            xvpi_addr_int <= device_addr;
            if (trans_byte = "11" and config_clk = '1'
                and write_done = '1') then -- done
              transfer_state <= STATUS_CHECK;
              flush_outpipe <= "00";
              xvpi_cs_n <= '1';
              xvpi_data <= "00000000"; -- Move SUCCESS here
            else
              xvpi_data <= check_byte_ddd;
            end if;

            -- Find start of bitstream (FF FF FF FF)
            -- before starting write
            check_byte_d <= check_byte; check_byte_dd <= check_byte_d;
            check_byte_ddd <= check_byte_dd;
            if (check_byte = "11111111"
                  and check_byte_d = "11111111"
                  and check_byte_dd = "11111111"
                  and check_byte_ddd = "11111111") then
              xvpi_wr_int <= '1';
              xvpi_cs_n <= '0';
            end if;

            -- Break TRANS_DATA into bytes for xvpi transfer
            if config_clk = '1' then
              trans_byte <= trans_byte + 1;
            end if;
            case trans_byte is
            when "00" =>
              write_done <= not transfer;
              trans_data_reg <= trans_data;
              check_byte <= trans_data(7 downto 0);
            when "01" =>
              check_byte <= trans_data_reg(15 downto 8);
            when "10" =>
              check_byte <= trans_data_reg(23 downto 16);
            when "11" =>
              -- hold control here -- waiting <= hold;
              check_byte <= trans_data_reg(31 downto 24);
            when others =>  -- added for workview simulation
              check_byte <= check_byte;
            end case;
          end if;

        when STATUS_CHECK =>

          xvpi_wr_int <= '0';
          xvpi_cs_n <= '1';
          if flush_outpipe = "10" then
            flush_outpipe <= "00";
            if (XVPI_Done = '1') then -- done = 1
              transfer_state <= SUCCESS;  -- go to IDLE instead.
            else
              status <= "11";
              transfer_state <= FAILED;
            end if;
          else  -- wait for valid xvpi_data_rd
            flush_outpipe <= flush_outpipe + 1;
          end if;

        when SUCCESS =>
        status <= "00"; -- Programming successful
        running <= '0';

          -- Send some extra cclks to start state machine
          xvpi_wr_int <= '1';
          trans_byte <= trans_byte + 1;
          if trans_byte = "11" then
            -- Turn off RW_N, CS_N, etc
            xvpi_wr_int <= '0';
            xvpi_data <=   (others => '0');
            xvpi_cs_n <= '1';
            xvpi_prog_n <= '1';
            transfer_state <= IDLE;

          end if;

        when FAILED =>
        status <= "11";  -- Programming Failed
        running <= '0';

          xvpi_wr_int <= '0';
          xvpi_data <=   (others => '0');
```

```
          xvpi_addr_int <= (others => '1');
          transfer_state <= IDLE;

      end case;
    end if;
    end process;

end STATE_MACHINE;
```

APPENDIX B

WIRED NETWORK DATA CLASSIFIER

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

-------------------------------------------------------------------------
-- User Classifier
--
--  Puts user data into channels
--  Input Data Format:
--    Control Byte in radio format
--    (CRC, Key Change, Radio Change, ARQ, Channel)
--  Data Size (LSB first)
--  Data
--
--  Scott Harper, Virginia Tech Dept of ECE, October 1, 2002
-------------------------------------------------------------------------

entity user_classifier is
  generic(DATA_WIDTH : integer);
  port(
    CLK    : in std_logic;
    RESET    : in std_logic;

    INPUT    : in std_logic_vector((DATA_WIDTH-1) downto 0);
    IN_VALID  : in std_logic;
    IN_ACK  : out std_logic;

    CH0_DATA    : out std_logic_vector((DATA_WIDTH-1) downto 0);
    CH0_VALID  : out std_logic;
    CH0_READY  : in std_logic;

    CH1_DATA    : out std_logic_vector((DATA_WIDTH-1) downto 0);
    CH1_VALID  : out std_logic;
    CH1_READY  : in std_logic;

    CH2_DATA    : out std_logic_vector((DATA_WIDTH-1) downto 0);
    CH2_VALID  : out std_logic;
    CH2_READY  : in std_logic;

    CH3_DATA    : out std_logic_vector((DATA_WIDTH-1) downto 0);
    CH3_VALID  : out std_logic;
    CH3_READY  : in std_logic;

    DEBUG : out std_logic_vector(3 downto 0)
    );
    constant START_MARK : std_logic_vector(7 downto 0) := X"CD";

end user_classifier;

architecture behavior of user_classifier is


type classify_state is (
  SEARCH,
  HEADER_SIZE1,
  HEADER_SIZE2,
  PASS_DATA,
  DELAY1,
  DELAY2

  );
signal sort_state : classify_state;

signal output, output_reg : std_logic_vector(7 downto 0);

signal checksum : std_logic_vector(15 downto 0);
signal header_check_pass : std_logic;
signal pkt_size : std_logic_vector(15 downto 0);
signal channel_select, channel_mask : std_logic_vector(3 downto 0);
signal CRC_Attached, Key_Change, Config_Change, Ack_Req : std_logic;
signal crc : std_logic_vector(15 downto 0);
signal crc_out : std_logic_vector(7 downto 0);
signal calc_crc, valid_tag, crc_data_valid : std_logic;
signal pkt_done : std_logic;
signal crc_reset : std_logic;
signal data_hold : std_logic_vector(7 downto 0);
signal out_valid, out_valid_reg : std_logic;
--signal IN_ACK_int : std_logic;
signal output_ready : std_logic;

type data_delay_type is array (4 downto 0) of std_logic_vector(DATA_WIDTH
downto 1);
signal control_data_delay : data_delay_type;

begin


CLASSIFY: process(RESET,CLK,IN_VALID)
  begin
  if RESET = '1' then
    pkt_done <= '0';
    sort_state <= SEARCH;
    Key_Change <= '0';
    Config_Change <= '0';
    CRC_Attached <= '0';
    Ack_Req <= '0';
    checksum <= (others => '0');
    pkt_size <= (others => '0');
    channel_select <= (others => '0');
    channel_mask <= (others => '0');
    data_hold <= (others => '0');
    header_check_pass <= '0';
    calc_crc <= '0';
    valid_tag <= '0';
    output <= START_MARK;
      out_valid <= '0';

  elsif CLK'event and CLK = '1' then
    case sort_state is

      when SEARCH => --  Wait for input data

        pkt_done <= '0';
        channel_mask <= (others => '0');
        valid_tag <= '0';
        out_valid <= '0';
        output <= INPUT;
        if IN_VALID = '1' then -- control byte
             channel_mask <= INPUT(3 downto 0);
               out_valid <= '1';
             sort_state <= DELAY1;
        end if;

      when DELAY1 => -- output fifo losing data when valid held high
         out_valid <= '0';
        sort_state <= HEADER_SIZE1;

      when HEADER_SIZE1 => -- 1st byte of size (LSB)

            output <= INPUT;
        if IN_VALID = '1' then
           out_valid <= '1';
          pkt_size(7 downto 0) <= INPUT;
          sort_state <= HEADER_SIZE2;
        end if;

      when HEADER_SIZE2 => -- 2nd byte of size

            output <= INPUT;
        if IN_VALID = '1' then
           out_valid <= '1';
          pkt_size(15 downto 8) <= INPUT;
          sort_state <= DELAY2;
            else
               out_valid <= '0';
        end if;

      when DELAY2 => -- kludge fifo problem
         out_valid <= '0';
        sort_state <= PASS_DATA;

      when PASS_DATA => -- pass data to correct channel

        if pkt_size = 0 then
          out_valid <= '0';
            pkt_done <= '1';
          sort_state <= SEARCH;
        else
```

```
          if IN_VALID = '1' then
            pkt_size <= pkt_size - 1;
            output <= INPUT;
            out_valid <= '1';
          else
             out_valid <= '0';
          end if;

        end if;

      end case;

  end if;
end process;

--REGISTER_SIGNALS: process(CLK, RESET, output)
--  begin
--  if (RESET = '1') then
--    output_reg <= (others => '0');
--    out_valid_reg <= '0';
--  elsif (CLK'event and CLK = '1') then
    output_reg <= output;
    out_valid_reg <= out_valid;
--  end if;
--end process;

  output_ready <= '1' when channel_mask = "0000" else
                  ((CH0_READY and channel_mask(0)) or
                  (CH1_READY and channel_mask(1)) or
                  (CH2_READY and channel_mask(2)) or
                  (CH3_READY and channel_mask(3)));

  IN_ACK <= output_ready and not RESET;

  CH0_DATA <= output_reg;
  CH0_VALID <= out_valid_reg when channel_mask(0) = '1' else '0';

  CH1_DATA <= output_reg;
  CH1_VALID <= out_valid_reg when channel_mask(1) = '1' else '0';

  CH2_DATA <= output_reg;
  CH2_VALID <= out_valid_reg when channel_mask(2) = '1' else '0';

  CH3_DATA <= output_reg;
  CH3_VALID <= out_valid_reg when channel_mask(3) = '1' else '0';

  DEBUG <=  -- channel_mask(3 downto 0);
        '1' & channel_mask(3 downto 1) when sort_state = SEARCH else
        '0' & channel_mask(3 downto 1);

end behavior;
```

APPENDIX C

WIRELESS DATA PACKETIZATION

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

-------------------------------------------------------------------------
-- Radio packetizer
--
--    Turns data into radio packet for transmit
--    Data Format:
--      Control Byte in radio format
--        (CRC, Key Change, Radio Change, ARQ, Channel)
--      Data Size (2 bytes - LSB first)
--      Data
--
-------------------------------------------------------------------------

entity packetizer is
  generic(DATA_WIDTH : integer);
  port(
    CLK    : in std_logic;
    RESET    : in std_logic;

    INPUT   : in std_logic_vector((DATA_WIDTH-1) downto 0);
    IN_VALID  : in std_logic;
    IN_ACK  : out std_logic;

    CH_DATA    : out std_logic_vector((DATA_WIDTH-1) downto 0);
    CH_VALID  : out std_logic;
    CH_READY  : in std_logic;

    DEBUG : out std_logic
    );

    constant START_MARK : std_logic_vector(7 downto 0) := X"CD";

end packetizer;

architecture behavior of packetizer is

component crc_ccitt
port (
  CLK : in std_logic;
  RESET : in std_logic;

  DATA : in std_logic_vector(7 downto 0);
  DATA_VALID : in std_logic;

  CRC : out std_logic_vector(15 downto 0)
  );
end component;


type packet_state is (
  SEARCH,
  HEADER_CONTROL,
  HEADER_SIZE1,
  HEADER_SIZE2,
  FINAL_CHECK1,
  FINAL_CHECK2,
  PASS_DATA,
  WRITE_CRC_RESULT,
  WRITE_CRC_RESULT2,
  DELAY1,
  DELAY2,
  DELAY3,
  DELAY4,
  DELAY5

  );
signal sort_state : packet_state;

signal output, output_reg : std_logic_vector(7 downto 0);

signal checksum : std_logic_vector(15 downto 0);
signal header_check_pass : std_logic;
signal pkt_size : std_logic_vector(15 downto 0);
signal channel_select, channel_mask : std_logic_vector(3 downto 0);
signal CRC_Attached, Key_Change, Config_Change, Ack_Req : std_logic;
signal crc : std_logic_vector(15 downto 0);
signal crc_out : std_logic_vector(7 downto 0);
signal calc_crc, valid_tag, crc_data_valid : std_logic;
signal pkt_done : std_logic;
signal crc_reset : std_logic;
signal data_hold : std_logic_vector(7 downto 0);
signal out_valid, out_valid_reg : std_logic;
signal IN_ACK_int : std_logic;
signal output_ready : std_logic;

type data_delay_type is array (4 downto 0) of std_logic_vector(DATA_WIDTH
downto 1);
signal control_data_delay : data_delay_type;

begin

CRC_CALC: crc_ccitt
port map(
  CLK => CLK,
  RESET => crc_reset,

  DATA => INPUT,
  DATA_VALID => crc_data_valid,
  CRC => crc
  );

crc_data_valid <= IN_VALID and calc_crc;
crc_reset <= RESET or pkt_done;

PACKETIZE: process(RESET,CLK,IN_VALID)
  begin
  if RESET = '1' then
    pkt_done <= '0';
    sort_state <= SEARCH;
    Key_Change <= '0';
    Config_Change <= '0';
    CRC_Attached <= '0';
    Ack_Req <= '0';
    checksum <= (others => '0');
    pkt_size <= (others => '0');
    channel_select <= (others => '0');
    channel_mask <= (others => '0');
    data_hold <= (others => '0');
    header_check_pass <= '0';
    calc_crc <= '0';
    valid_tag <= '0';
    output <= START_MARK;
      out_valid <= '0';
    IN_ACK_int <= '1';

  elsif CLK'event and CLK = '1' then
    case sort_state is

      when SEARCH => --  Wait for input data, then write start mark

        pkt_done <= '0';
        calc_crc <= '0';
        valid_tag <= '0';
          output <= START_MARK;
        out_valid <= '0';
        if IN_VALID = '1' then
             checksum <= INPUT & START_MARK;
             CRC_Attached <= INPUT(7);
             Key_Change <= INPUT(6);
             Config_Change <= INPUT(5);
             Ack_Req <= INPUT(4);
             data_hold <= INPUT;
               out_valid <= '1';
             sort_state <= DELAY1;
             IN_ACK_int <= '0';
        end if;

      when DELAY1 => -- output fifo losing data when valid held high
         out_valid <= '0';
        sort_state <= HEADER_CONTROL;

      when HEADER_CONTROL => -- write packet control info

         out_valid <= '1';
        output <= data_hold;
```

```
          IN_ACK_int <= '1';
          sort_state <= HEADER_SIZE1;

        when HEADER_SIZE1 => -- 1st byte of size (LSB)

              output <= INPUT;
          if IN_VALID = '1' then
             out_valid <= '1';
            checksum <= checksum + ("00000000" & INPUT);
            pkt_size(7 downto 0) <= INPUT;
            sort_state <= HEADER_SIZE2;
          else
             out_valid <= '0';
          end if;

        when HEADER_SIZE2 => -- 2nd byte of size

              output <= input;
          if IN_VALID = '1' then
             out_valid <= '1';
            checksum <= checksum + (INPUT & "00000000");
            pkt_size(15 downto 8) <= INPUT;
            IN_ACK_int <= '0';
            sort_state <= DELAY2;
              else
                 out_valid <= '0';
          end if;

        when DELAY2 => -- kludge fifo problem
           out_valid <= '0';
          sort_state <= FINAL_CHECK1;

        when FINAL_CHECK1 => -- compute header checksum

              out_valid <= '1';
           output <= not checksum(7 downto 0);
              out_valid <= '1';
          sort_state <= DELAY3;

        when DELAY3 => -- kludge fifo problem
           out_valid <= '0';
          sort_state <= FINAL_CHECK2;

        when FINAL_CHECK2 => -- compute header checksum

           out_valid <= '1';
          output <= not checksum(15 downto 8);
          IN_ACK_int <= '1';
          sort_state <= PASS_DATA;

        when PASS_DATA => -- pass data to correct channel

            calc_crc <= '1';
          if pkt_size = 0 then
            out_valid <= '0';
             calc_crc <= '0';
            if CRC_Attached = '1' then
```

```
             IN_ACK_int <= '0';
            sort_state <= DELAY4;
          else
              pkt_done <= '1';
            IN_ACK_int <= '1';
            sort_state <= SEARCH;
          end if;
        else
          if IN_VALID = '1' then
            pkt_size <= pkt_size - 1;
            output <= INPUT;
            out_valid <= '1';
          else
             out_valid <= '0';
          end if;

        end if;

      when DELAY4 => -- kludge fifo problem
            output <= CRC(7 downto 0);
            out_valid <= '1';
        sort_state <= DELAY5;

      when DELAY5 => -- kludge fifo problem
         out_valid <= '0';
        sort_state <= WRITE_CRC_RESULT;

      when WRITE_CRC_RESULT => --write CRC pass/fail data to channel fifo

         out_valid <= '1';
        output <= CRC(15 downto 8);
        sort_state <= WRITE_CRC_RESULT2;

      when WRITE_CRC_RESULT2 => -- Done

        pkt_done <= '1';
            out_valid <= '0';
        IN_ACK_INT <= '1';
        sort_state <= SEARCH;

      end case;

  end if;
end process;

--REGISTER_SIGNALS: process(CLK, RESET, output)
--  begin
--  if (RESET = '1') then
--    output_reg <= (others => '0');
--    out_valid_reg <= '0';
--  elsif (CLK'event and CLK = '1') then
    output_reg <= output;
    out_valid_reg <= out_valid;
--  end if;
--end process;

  output_ready <= CH_READY;
```

```
   IN_ACK <= IN_ACK_int and output_ready and not RESET;

  CH_DATA <= output_reg;
  CH_VALID <= out_valid_reg;

  DEBUG <= '1' when sort_state = SEARCH else
        '0';

end behavior;
```

APPENDIX D

RADIO DATA OUTPUT ARBITER

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

-------------------------------------------------------------------------
-- Arbiter
--
--    Arbitrates output channels to UART
--
--  Scott Harper, Virginia Tech Dept of ECE, June 28, 2002
-------------------------------------------------------------------------

entity arbiter is
  generic(DATA_WIDTH : integer);
  port(
    CLK    : in std_logic;
    RESET    : in std_logic;

    OUTPUT    : out std_logic_vector((DATA_WIDTH-1) downto 0);
    OUT_VALID  : out std_logic;
    OUT_ACK    : in std_logic;

    CH0_DATA    : in std_logic_vector((DATA_WIDTH-1) downto 0);
    CH0_VALID  : in std_logic;
    CH0_READY  : out std_logic;

    CH1_DATA    : in std_logic_vector((DATA_WIDTH-1) downto 0);
    CH1_VALID  : in std_logic;
    CH1_READY  : out std_logic;

    CH2_DATA    : in std_logic_vector((DATA_WIDTH-1) downto 0);
    CH2_VALID  : in std_logic;
    CH2_READY  : out std_logic;

    CH3_DATA    : in std_logic_vector((DATA_WIDTH-1) downto 0);
    CH3_VALID  : in std_logic;
    CH3_READY  : out std_logic;

    DEBUG : out std_logic_vector(3 downto 0)
    );

end arbiter;

architecture behavior of arbiter is
```

```
type arbitration_state is (
  SEARCH,
  CONTROL,
  HEADER_SIZE1,
  HEADER_SIZE2,
  PASS_DATA,
  FLUSH,
  DELAY0,
  DELAY1,
  DELAY2,
  DELAY3,
  DELAY4
  );
signal arbiter_state : arbitration_state;
signal pkt_size : std_logic_vector(15 downto 0);
signal channel_mask : std_logic_vector(3 downto 0);
signal ready_int : std_logic;
signal input, data_reg : std_logic_vector(7 downto 0);
signal in_valid : std_logic;
signal ch_valid : std_logic_vector(3 downto 0);
signal crc_attached : std_logic;

begin


PACKET_PASS: process(RESET,CLK,in_valid)
  begin
  if RESET = '1' then
    arbiter_state <= SEARCH;
    pkt_size <= (others => '0');
    channel_mask <= "0001";
      OUT_VALID <= '0';
    data_reg <= (others => '0');
    ready_int <= '1';
    crc_attached <= '0';

  elsif CLK'event and CLK = '1' then
     data_reg <= INPUT;

    if (OUT_ACK = '1') then
    case arbiter_state is

      when SEARCH => --  Wait for input data

        if (in_valid = '1') and (INPUT = X"CD") then -- start byte
               OUT_VALID <= '1';
             arbiter_state <= DELAY0;
             ready_int <= '0';
        else
          channel_mask <= channel_mask(0) & channel_mask(3)
                          & channel_mask(2) & channel_mask(1);
          OUT_VALID <= '0';
        end if;

      when DELAY0 => -- output fifo losing data when valid held high
         OUT_VALID <= '0';
        arbiter_state <= CONTROL;
        ready_int <= '1';

      when CONTROL => -- Control Byte

        if in_valid = '1' then
           OUT_VALID <= '1';
          arbiter_state <= DELAY1;
          crc_attached <= INPUT(7);
          ready_int <= '0';
        else
           OUT_VALID <= '0';
        end if;

      when DELAY1 => -- output fifo losing data when valid held high
         OUT_VALID <= '0';
        arbiter_state <= HEADER_SIZE1;
        ready_int <= '1';

      when HEADER_SIZE1 => -- 1st byte of size (LSB)

        if in_valid = '1' then
           OUT_VALID <= '1';
          pkt_size(7 downto 0) <= INPUT;
          arbiter_state <= DELAY2;
          ready_int <= '0';
        else
           OUT_VALID <= '0';
        end if;

      when DELAY2 => -- output fifo losing data when valid held high
         OUT_VALID <= '0';
        arbiter_state <= HEADER_SIZE2;
        ready_int <= '1';


      when HEADER_SIZE2 => -- 2nd byte of size

        if in_valid = '1' then
           OUT_VALID <= '1';
          pkt_size(15 downto 8) <= INPUT;
          arbiter_state <= DELAY3;
          ready_int <= '0';
            else
               OUT_VALID <= '0';
        end if;

      when DELAY3 => -- kludge fifo problem
        if (crc_attached = '1' ) then
          pkt_size <= pkt_size + 4;
        else
          pkt_size <= pkt_size + 2;
        end if;
         OUT_VALID <= '0';
        arbiter_state <= PASS_DATA;
        ready_int <= '1';

      when PASS_DATA => -- pass data to correct channel
```

```
        if pkt_size = 0 then
          arbiter_state <= FLUSH;
          out_valid <= '0';
        else
          if in_valid = '1' then
            pkt_size <= pkt_size - 1;
            OUT_VALID <= '1';
          else
            OUT_VALID <= '0';
          end if;
        end if;

      when FLUSH => -- flush buffered data
         OUT_VALID <= '0';
         arbiter_state <= DELAY4;
         ready_int <= '0';

      when DELAY4 =>
        OUT_VALID <= '0';
        arbiter_state <= SEARCH;
        ready_int <= '1';

      end case;
    end if;
  end if;
end process;

in_valid <= '0' when (channel_mask and ch_valid) = "0000" else '1';
ch_valid <= CH3_VALID & CH2_VALID & CH1_VALID & CH0_VALID;

input <=  CH0_DATA when channel_mask(0) = '1' else
          CH1_DATA when channel_mask(1) = '1' else
          CH2_DATA when channel_mask(2) = '1' else
          CH3_DATA;

OUTPUT <= data_reg;
          -- CH0_DATA when channel_mask(0) = '1' else
          -- CH1_DATA when channel_mask(1) = '1' else
          -- CH2_DATA when channel_mask(2) = '1' else
          -- CH3_DATA;

CH0_READY <= OUT_ACK and ready_int and channel_mask(0);
CH1_READY <= OUT_ACK and ready_int and channel_mask(1);
CH2_READY <= OUT_ACK and ready_int and channel_mask(2);
CH3_READY <= OUT_ACK and ready_int and channel_mask(3);

DEBUG <=  -- channel_mask(3 downto 0);
      '1' & channel_mask(3 downto 1) when arbiter_state = SEARCH else
      '0' & channel_mask(3 downto 1);

end behavior;
```

APPENDIX E

RADIO PACKET CLASSIFIER

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

-------------------------------------------------------------------------
-- Radio packet classifier
--
--      Strips packet header, routes data, and checks CRC
--
--  Scott Harper, Virginia Tech Dept of ECE, June 15, 2002
-------------------------------------------------------------------------

entity classifier is
   generic(DATA_WIDTH : integer);
   port(
    CLK      : in std_logic;
    RESET      : in std_logic;

    INPUT      : in std_logic_vector(DATA_WIDTH downto 1);
    IN_VALID   : in std_logic;

    CH0_DATA    : out std_logic_vector(DATA_WIDTH downto 1);
    CH0_VALID  : out std_logic;

    CH1_DATA    : out std_logic_vector(DATA_WIDTH downto 1);
    CH1_VALID  : out std_logic;

    CH2_DATA    : out std_logic_vector(DATA_WIDTH downto 1);
    CH2_VALID  : out std_logic;

    CH3_DATA    : out std_logic_vector(DATA_WIDTH downto 1);
    CH3_VALID  : out std_logic;

    CONTROL_DATA  : out std_logic_vector(DATA_WIDTH downto 1);
    CONTROL_VALID  : out std_logic;

    ARQ_SEND  : out std_logic;

    CHANGE_CONFIG : out std_logic;
    CHANGE_KEY : out std_logic;
    END_OF_PKT  : out std_logic_vector(3 downto 0);

    DEBUG : out std_logic_vector(3 downto 0)
    );
    constant START_MARK : std_logic_vector(7 downto 0) := "11001101";

end classifier;

architecture behavior of classifier is

component crc_ccitt
port (
  CLK : in std_logic;
  RESET : in std_logic;

  DATA : in std_logic_vector(7 downto 0);
  DATA_VALID : in std_logic;

  CRC : out std_logic_vector(15 downto 0)
  );
end component;


type classify_state is (
  SEARCH,
  HEADER_CONTROL,
  HEADER_SIZE1,
  HEADER_SIZE2,
  FINAL_CHECK1,
  FINAL_CHECK2,
  PASS_DATA,
  WRITE_CRC_RESULT,
  WRITE_CRC_RESULT2,
  WRITE_CRC_RESULT3

  );
signal sort_state : classify_state;

signal checksum : std_logic_vector(15 downto 0);
signal header_check_pass : std_logic;
signal pkt_size : std_logic_vector(15 downto 0);
signal channel_select, channel_mask : std_logic_vector(4 downto 0);
signal CRC_Attached, Key_Change, Config_Change, Ack_Req : std_logic;
signal control_write, write_crc_upper, write_crc_clk, write_crc :
std_logic;
signal crc_result : std_logic_vector(15 downto 0);
signal crc : std_logic_vector(15 downto 0);
signal crc_out : std_logic_vector(7 downto 0);
signal valid_pulse, calc_crc, valid_tag, crc_data_valid : std_logic;
signal pkt_done : std_logic;
signal ack_state, ack_rec : std_logic;
signal crc_reset : std_logic;

type data_delay_type is array (4 downto 0) of std_logic_vector(DATA_WIDTH
downto 1);
signal control_data_delay : data_delay_type;
signal control_write_delay : std_logic_vector(4 downto 0);

begin

CRC_CALC: crc_ccitt
```

```
port map(
  CLK => CLK,
  RESET => crc_reset,

  DATA => INPUT,
  DATA_VALID => crc_data_valid,

  CRC => crc
  );

crc_data_valid <= VALID_PULSE and calc_crc;
crc_reset <= RESET or pkt_done;
valid_pulse <= IN_VALID;

CLASSIFY: process(RESET,CLK,valid_pulse)
  begin
  if RESET = '1' then
    pkt_done <= '0';
    sort_state <= SEARCH;
    Key_Change <= '0';
    Config_Change <= '0';
    CRC_Attached <= '0';
    Ack_Req <= '0';
    ARQ_SEND <= '0';
    checksum <= (others => '0');
    pkt_size <= (others => '0');
    channel_select <= (others => '0');
    channel_mask <= (others => '0');
    control_write <= '0';
    crc_result <= (others => '0');
    write_crc_upper <= '0';
    write_crc_clk <= '0';
    header_check_pass <= '0';
    calc_crc <= '0';
    valid_tag <= '0';

  elsif CLK'event and CLK = '1' then
    case sort_state is

      when SEARCH => --  Search for 0xCD packet start marker

        pkt_done <= '0';
        channel_mask <= (others => '0');
        header_check_pass <= '0';
        write_crc_clk <= '0';
        write_crc_upper <= '0';
        calc_crc <= '0';
        valid_tag <= '0';
        crc_result <= (others => '0');
        ack_rec <= '0';
        if valid_pulse = '1' and INPUT = START_MARK then
          -- checksum <= "00000000" & INPUT;
          control_write <= '1';
          sort_state <= HEADER_CONTROL;
        end if;

      when HEADER_CONTROL => -- parse header control info
        if valid_pulse = '1' then
           if INPUT = X"FF" then
              ack_rec <= '1';
              sort_state <= SEARCH;
           else
             checksum <= INPUT & START_MARK;
             CRC_Attached <= INPUT(8);
             Key_Change <= INPUT(7);
             Config_Change <= INPUT(6);
             Ack_Req <= INPUT(5);
             channel_select <= INPUT(5 downto 1);
             sort_state <= HEADER_SIZE1;
          end if;
        end if;

      when HEADER_SIZE1 => -- 1st byte of size (MSB)

        if valid_pulse = '1' then
          checksum <= checksum + ("00000000" & INPUT);
          pkt_size(7 downto 0) <= INPUT;
          sort_state <= HEADER_SIZE2;
        end if;

      when HEADER_SIZE2 => -- 2nd byte of size

        if valid_pulse = '1' then
          control_write <= '0';
          checksum <= checksum + (INPUT & "00000000");
          pkt_size(15 downto 8) <= INPUT;
          sort_state <= FINAL_CHECK1;
        end if;

      when FINAL_CHECK1 => -- compute header checksum

        if valid_pulse = '1' then
          checksum <= checksum + ("00000000" & INPUT);
          if CRC_Attached = '1' then pkt_size <= pkt_size + 2; end if;
          sort_state <= FINAL_CHECK2;
        end if;

      when FINAL_CHECK2 => -- compute header checksum

        if valid_pulse = '1' then
          if checksum + (INPUT & "00000000") = "1111111111111111" then
            sort_state <= PASS_DATA;
            channel_mask <= channel_select;
            header_check_pass <= '1';
            calc_crc <= '1';
          else
            sort_state <= SEARCH;
          end if;
        end if;

      when PASS_DATA => -- pass data to correct channel

        if pkt_size = 0 then
```

```
          sort_state <= WRITE_CRC_RESULT;
          if CRC_Attached = '1' then crc_result <= CRC; end if;
          calc_crc <= '0';
          valid_tag <= '1';
        else
          if valid_pulse = '1' then pkt_size <= pkt_size - 1; end if;
        end if;

      when WRITE_CRC_RESULT => --write CRC pass/fail data to channel fifo

        write_crc_clk <= '1';
        if (CRC = "0000000000000000") and (Ack_Req = '1') then
          ARQ_SEND <= '1';
        end if;
        sort_state <= WRITE_CRC_RESULT2;

      when WRITE_CRC_RESULT2 => -- write CRC p/f data to channel fifo

        ARQ_SEND <= '0';
        write_crc_clk <= '0';
        write_crc_upper <= '1';
        sort_state <= WRITE_CRC_RESULT3;

      when WRITE_CRC_RESULT3 => -- write CRC p/f data to channel fifo

        write_crc_clk <= '1';
        pkt_done <= '1';
        sort_state <= SEARCH;

      end case;

  end if;
end process;


CONTROL_DELAY: process(CLK, RESET, valid_pulse)
  -- Delay write of control data until header checksum result is known
  begin
    if RESET = '1' then
      control_data_delay(0) <= (others => '0');
      control_data_delay(1) <= (others => '0');
      control_data_delay(2) <= (others => '0');
      control_data_delay(3) <= (others => '0');
      control_data_delay(4) <= (others => '0');
      control_write_delay <= (others => '0');
    elsif (CLK'event and CLK = '1') then
      if valid_pulse = '1' then

      control_data_delay(4) <= INPUT;
      control_data_delay(3) <= control_data_delay(4);
      control_data_delay(2) <= control_data_delay(3);
      control_data_delay(1) <= control_data_delay(2);
      control_data_delay(0) <= control_data_delay(1);

      control_write_delay(4) <= control_write;
      control_write_delay(3) <= control_write_delay(4);
      control_write_delay(2) <= control_write_delay(3);
      control_write_delay(1) <= control_write_delay(2);
      control_write_delay(0) <= control_write_delay(1);

      end if;

    end if;
end process;

  crc_out <= crc_result(15 downto 8) when write_crc_upper = '1' else
        crc_result(7 downto 0);

  write_crc <= write_crc_clk;

  CH0_DATA <= INPUT;
  CH0_VALID <= valid_pulse when channel_mask(0) = '1' else '0';
  END_OF_PKT(0) <= pkt_done and channel_mask(0);

  CH1_DATA <= INPUT;
  CH1_VALID <= valid_pulse when channel_mask(1) = '1' else '0';
  END_OF_PKT(1) <= pkt_done and channel_mask(1);

  CH2_DATA <= INPUT;
  CH2_VALID <= valid_pulse when channel_mask(2) = '1' else '0';
  END_OF_PKT(2) <= pkt_done and channel_mask(2);

  CH3_DATA <= INPUT;
  CH3_VALID <= valid_pulse when channel_mask(3) = '1' else '0';
  END_OF_PKT(3) <= pkt_done and channel_mask(3);

  -- Note this control write scheme requires the size of regular packet
data
  -- to be at least the same as the header size so that valid pulses are
produced.
  CONTROL_DATA <= crc_out when valid_tag = '1' else
control_data_delay(0);
  CONTROL_VALID <= (header_check_pass and valid_pulse and
control_write_delay(0)) or write_crc;

  CHANGE_KEY <= Key_Change;
  CHANGE_CONFIG <= Config_Change;

  DEBUG <=  -- channel_mask(3 downto 0);
        '1' & channel_mask(3 downto 1) when sort_state = SEARCH else
        '0' & channel_mask(3 downto 1);

end behavior;
```

“Never worry about theory as long as the machinery does what it’s supposed to do.”
- Robert Heinlein

VITA

Scott Harper graduated in 1983 from Chippewa Falls Senior High School in Chippewa Falls, Wisconsin, with his eye on Electrical Engineering. After brief stints at the University of Wisconsin campuses at Madison and Eau Claire, he received his Bachelor of Science degree in Electrical Engineering from the University of Wisconsin at Platteville in 1989. Seeking higher education, he traveled east to the Virginia Polytechnic Institute and State University in Blacksburg, Virginia. There, his investigations into the design of a hardware interface for a parallel network system lead to a Master of Science degree in Electrical Engineering in 1994. Recent publications include "Reconfigurable Object Detection in FLIR Image Sequences" (IEEE Symposium on Field Programmable Custom Computing Machines, pp. 284-285, Napa, California, April 2002), "A Stream-based Reconfigurable Router Prototype” (IEEE International Conference on Communications, pp. 581-585, Vancouver, B.C., June 1999), and "A Stream-Based Configurable Computing Radio Testbed" (IEEE Symposium on FPGAs for Custom Configurable Computing Machines, pp. 40-47, Napa, California, April 1998).