NORTHWESTERN UNIVERSITY

Compiler and Architectural Approaches to Software Protection and Security

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Electrical and Computer Engineering

By

Joseph Anthony Zambreno

EVANSTON, ILLINOIS

June 2006

ABSTRACT

Compiler and Architectural Approaches to

Software Protection and Security

Joseph Anthony Zambreno

One of the key problems facing the computer industry today is ensuring the integrity of end-user applications and data. Researchers in the relatively new field of software protection investigate the development and evaluation of controls that prevent the unauthorized modification or use of system software. While many previously developed protection schemes have provided a strong level of security, their overall effectiveness has been hindered by a lack of transparency to the user in terms of performance overhead. Other approaches take to the opposite extreme and sacrifice security for the sake of this transparency. In this work we present an architecture for software protection that provides for a high level of both security and user transparency by utilizing Field Programmable Gate Array (FPGA) technology as the main protection mechanism and a hardware/software co-design methodology. We demonstrate that by relying on FPGA technology, this approach can accelerate the execution of programs in a cryptographic environment, while

maintaining the flexibility through reprogramming to carry out any compiler-driven protections that may be application-specific. Results demonstrate that this framework can be the successful basis for the development of applications that meet a wide range of security and performance requirements.

Acknowledgements

I would first like to express thanks to my advisor, Professor Alok Choudhary, not only for motivating and directing this research, but for providing me with ample opportunities to expand my role as a graduate student. Also, much credit is due to Professors Bhagirath Narahari and Rahul Simha from The George Washington University, whose clarity of thinking has been especially helpful in analyzing my results and in supplying sincere feedback. I would also like to thank Professors Gokhan Memik, Seda Memik, and Russ Joseph for serving on my Final Examination committee. Finally, I would be remiss if I didn't mention the support of my wife Mary, who has encouraged me at every step of this journey.

This work was funded in part by the National Science Foundation (NSF) under grants CCR-0325207, CNS-0551639, IIS-0536994, by the Air Force Office of Scientific Research (AFOSR), and also by an NSF graduate research fellowship.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

Threats to a particular piece of software can originate from a variety of sources. A substantial problem from an economic perspective is the unauthorized copying and redistribution of applications, otherwise known as *software piracy*. Although the actual damage sustained from the piracy of software is certainly a debatable matter, some industry watchdog groups have estimated that software firms in 2002 lost as much as $2 billion in North American sales alone [**44**]. A threat that presents a much more direct harm to the end-user is *software tampering*, whereby a hacker maliciously modifies and redistributes code in order to cause large-scale disruptions in software systems or to gain access to critical information. For these reasons, *software protection* is considered one of the more important unresolved research challenges in security today [**22**]. In general, any software protection infrastructure should include 1) a method of limiting an attacker's ability to understand the higher level semantics of an application given a low-level (usually binary) representation, and 2) a system of checks that make it suitably difficult to modify the code at that low level. When used in combination, these two features can be extremely effective in preventing the circumvention of software authorization mechanisms.

While software protection is an important issue for desktop and servers, the fact that over 97% of all processors are embedded processors [**40**] shows the importance of protecting software on embedded systems. Imagine an attack on a key networked microcontroller used in transportation systems: the resulting tampered executables can cause large-scale

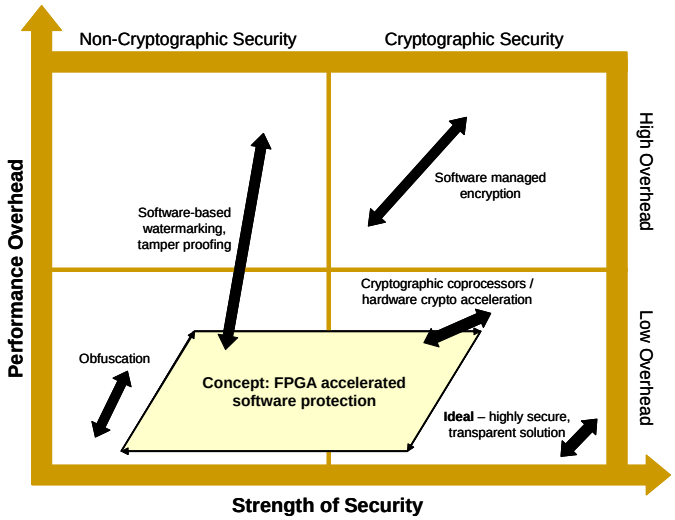

Figure 1.1. Performance and security strength of various software protection approaches

disruption of mechanical systems. Such an attack can be easily replicated because embedded processors are so numerous.

Current approaches to software protection can be categorized both by the strength of security provided and the performance overhead when compared to an unprotected environment (see Fig. 1.1). Two distinct categories emerge from this depiction: on one end of the security spectrum are the solely compiler-based techniques that implement both static and dynamic code validation through the insertion of objects into the generated executable; on the other end are the somewhat more radical methods that encrypt all instructions and data and that often require the processor to be architected with cryptographic hardware. Both of these methods have some practical limitations. The software-based techniques will only hinder an attacker, since tools can be built to identify and circumvent the protective checks. On the other hand, the cryptographic hardware

approaches, while inherently more secure, are limited in the sense that their practical implementation requires a wholesale commitment to a custom processor technology. More background on these software protection schemes can be found in Chapter 3.

Also, as can be inferred from Fig. 1.1, software protection is not an all-or-nothing concept, as opposed to other aspects of computer security where the effectiveness of an approach depends on its mathematical intractability; the resultant performance overhead is often not a key design consideration of these systems. In contrast, performance is equally important to the strength of security when protecting user applications, as there is generally some level of user control over the system as a whole. Consequently, any software protection scheme that is burdensome from a performance perspective will likely be turned off or routed around.

Field Programmable Gate Arrays (FPGAs) are hardware resources that combine various amounts of user-defined digital logic with customizable interconnect and I/O. A key feature of FPGAs is that their functionality can be reconfigured on multiple occasions, allowing for changes in the design to be implemented after the initial time of development. FPGAs have become an increasingly popular choice among architects in fields such as multimedia processing or cryptography - this has been attributed to the fact that the design process is much more streamlined than that for ASICs, as FPGAs are a fixed hardware target.

In this paper we present a high-performance architecture for software protection that uses this type of reconfigurable technology. By utilizing FPGAs as the main protection mechanism, this approach is able to merge the application tunability of the compiler-based methods with the additional security that comes with a hardware implementation.

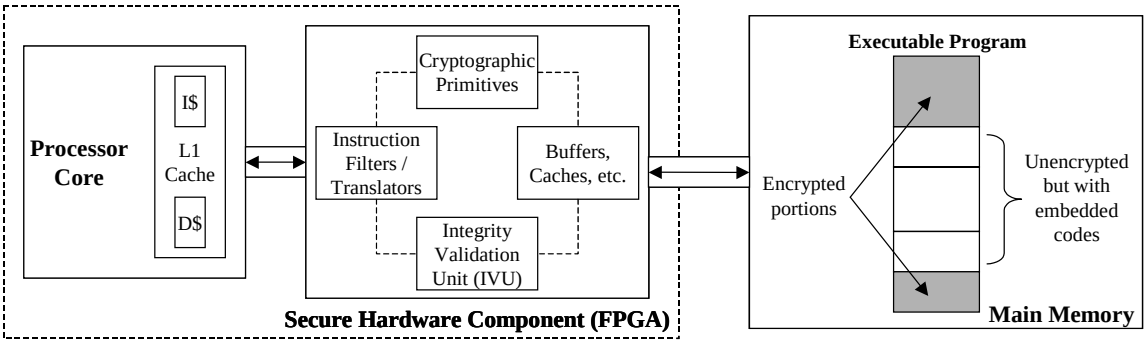

Figure 1.2. Conceptual view

As can be seen in Fig. 1.2, our proposed method works by supplementing a standard processor with an FPGA-based Integrity Validation Unit (IVU) that sits between the highest level of on-chip cache and main memory. This IVU is capable of performing fast decryption similar to any other hardware accelerated cryptographic coprocessing scheme, but more importantly the IVU also has the ability to recognize and certify binary messages hidden inside regular unencrypted instructions. Consequently our approach is completely complementary to current code restructuring techniques found in the software protection literature, with the added benefit that as the run-time code checking can be performed entirely in hardware it will be considerably more efficient. Our experiments show, that for most of our benchmarks, the inclusion of the FPGA within the instruction stream incurs a performance penalty of less than 20%, and that this number can be greatly improved upon with the utilization of unreserved reconfigurable resources for architectural optimizations, such as buffering and prefetching.

This approach exhibits the following advantages:

- It simultaneously addresses the four main types of attacks on software integrity: *code understanding*, *code tampering*, *data tampering*, and *authorization circumvention*
- The compiler's knowledge of program structure, its execution profile, and the programmability of the FPGA allow for tuning the security and performance for individual applications
- The approach is complementary to several software-based instruction validation techniques proposed recently [**15, 20, 41**];
- The hardware required in this scheme is simple and fast
- The use of FPGAs minimizes additional hardware design and is applicable to a large number of commercial processor platforms
- This processor/FPGA architecture is well-suited for future designs that utilize SoC technology

The remainder of this thesis is organized as follows. Chapter 2 sets the backdrop of our research, both illustrating the threat model under which we apply our approach and making the argument for the level of security that is provided. In Chapter 3 we provide additional background into the field of software security, with a review of some of the more commonly-used defensive techniques. Chapter 4 provides more details about our architecture, by illustrating how an FPGA situated in the instruction stream can be utilized to ensure software integrity. In this section we also provide an introduction to some custom compiler techniques that are well suited to such an architecture. In Chapter 5 we examine the performance implications of our approach, first by explicitly quantifying the security/performance tradeoff and then by performing experimental analysis. This

chapter also investigates the performance benefits of some hardware and software optimizations that can be applied within the experimental framework. In Chapter 6 we present preliminary results regarding the design and implementation of a compiler module and reconfigurable architecture that protects against several classes of program flow attacks. Because of the key role that cryptography plays in many of our proposed secure systems, we dedicate the entirety of Chapter 7 to an investigation of some architectural optimizations for a class of symmetric block ciphers. Finally, in Chapter 8 conclusions are presented, with a discussion of future optimizations and analysis techniques that are currently in development.

CHAPTER 2

Threats Considered

Consider a typical von Neumann architecture with a CPU and main memory (RAM). In the standard model, a program consisting of instructions and data is placed in RAM by a loader or operating system. Then, the CPU fetches instructions and executes them. Apart from the complexity introduced by multiprocessors and threads, this basic model applies to almost any computing system today including desktops, servers and small embedded devices.

However, from a security point of view, the execution of an application is far from safe. An attacker with access to the machine can examine the program (information leakage) and actively interfere with the execution (disruption) while also accumulating information useful in attacks on similar systems.

2.1. Encrypted Execution and Data (EED) Systems

The starting point for our proposed work is a recent body of work that has proposed building computing platforms with encrypted execution. We refer to these as EED (Encrypted Execution and Data) platforms. In these platforms, the executable and application data are encrypted. The processor or supporting hardware is assumed to have a key. The overall goals are to prevent leakage of information, to prevent tampering and to prevent disruption. For the highest degree of security, both instructions and data will

need to be encrypted using well-established encryption techniques. It should be noted that full-fledged EED platforms are still in their infancy.

The basis for our proposed work is the following:

- EED platforms, while undoubtedly more secure than the standard von Neumann model, are nonetheless still vulnerable to attacks that do not need decryption. That is, the attacker can find vulnerabilities without needing to decrypt and understand the software. We will refer to these attacks as EED attacks.
- Because attackers are presumed to be sophisticated, neither the RAM nor the CPU can be fully trusted. This situation also arises when RAM and CPU manufacturing is sub-contracted to third parties whose designs or cores cannot be easily verified.
- FPGAs have proved adept at solving performance-related problems in many computing platforms. As a result, tested FPGAs are commercially available for a variety of processors. Our approach exploits the combination of the programmability of FPGAs with the inherent additional security involved with computing directly in hardware to address EED attacks.
- A key part of our exploiting FPGAs involves the use of compiler technology to analyze program structure to enable best use of the FPGA, to help address key management and to increase performance. Consequently our approach allows for a level of security that is tunable to an individual application.

In a nutshell, the addition of the FPGA to the von Neumann model results in a new platform to sustain EED attacks.

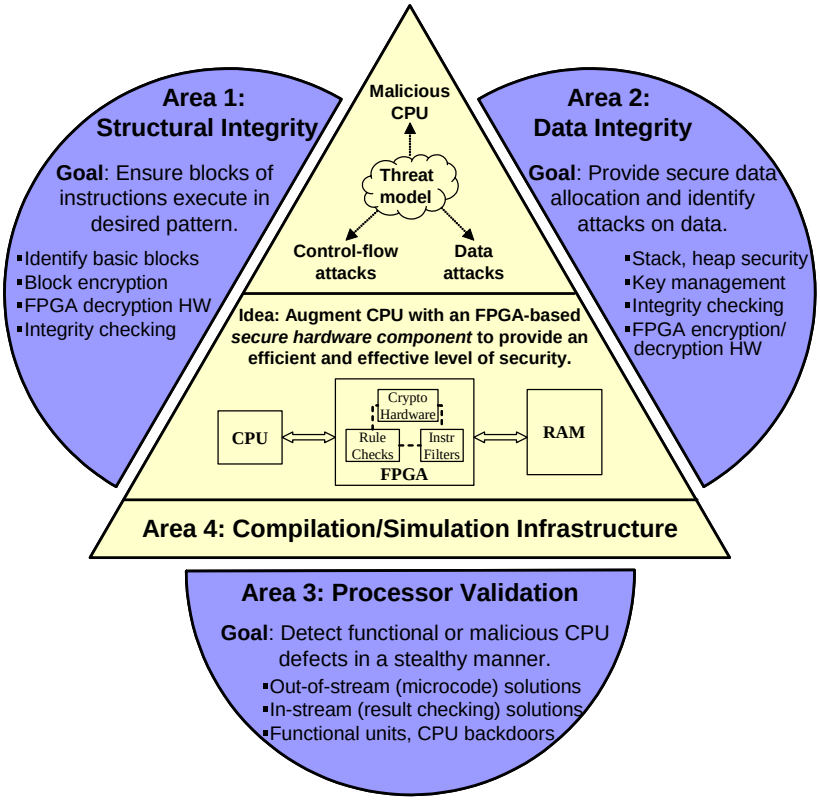

Figure 2.1. Research focus areas as they relate to the conceptual view

The contributions of our work can be categorized into four areas as seen in Fig. 2.1. We use the term *structural integrity* to refer to the proper execution path of a program when the data is assumed to be correct. Since an EED attacker can alter the control flow without decryption or even touching the data, we refer to such an attack as a structural EED attack. This first area of contributions is shown as Area 1 in the figure, in which we use a compiler-FPGA approach to address structural attacks.

The second area of contribution arises from considering EED attacks on *data integrity*. Our contribution to this second area is the use of the compiler-FPGA approach to provide key management techniques for data and data integrity techniques for runtime data.

The third area, *processor validation*, arises from considering a maliciously inserted processor instrumented to allow the attacker control over functional operations. Our approach is to have the compiler create code with processor-validation meta-instructions that are then interpreted in the FPGA to validate desired processor operations. Finally, we have developed a compiler-FPGA infrastructure for the purpose of implementing and testing our ideas. This infrastructure targets a modern processor (ARM family) and compiler (gcc).

2.2. Vulnerabilities in EED Systems

As mentioned in Chapter 1, research in the field of software protection has seen its motivation arrive from two different directions. On one side are vendors of various types of electronic media - their main concern is the unauthorized use or copying of their product. On the other side are those end users whose main interest is in protecting personal and corporate systems from outside interference. While the goals may be different, the approaches used by hackers in avoiding Digital Rights Management (DRM) schemes are often quite similar to those used by malicious crackers in attacking web servers and other unsecured applications. Hackers around the world know that the first step in attacking a software system is to first understand the software through the use of a debugger or other tracing utilities, and then to tamper with the software to enable a variety of exploits. Common means of exploiting software include buffer overflows, malformed `printf()` statements, and macro viruses. A detailed illustration of an exploitable buffer overflow is provided in the following section.

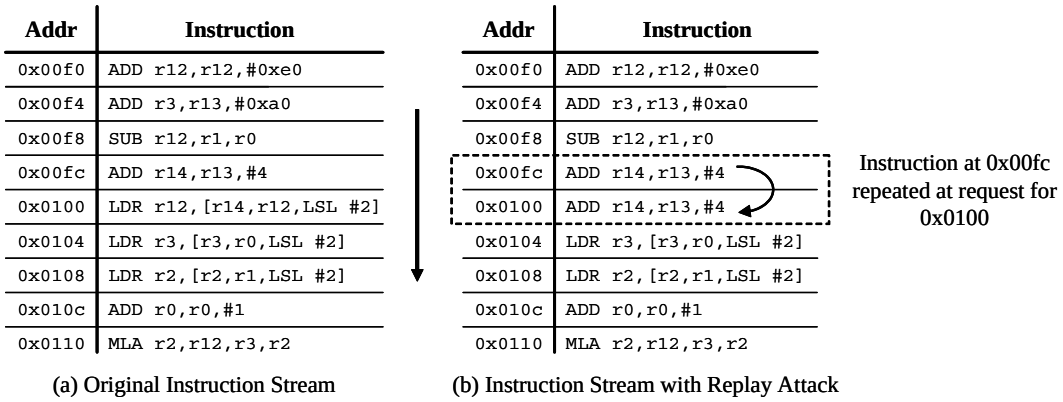

Figure 2.2. A sample instruction stream that is targeted with a replay attack

Consider a sophisticated attacker who has physical control of an EED computing platform. In a resourceful laboratory, the attacker can control the various data, address and control lines on the board and will have access to the designs of well-known commercial chips. Using this information, the attacker can actively interfere with the handshaking protocol between CPU and RAM, can insert data into RAM or even switch between the actual RAM and an attacker's RAM during execution.

We will assume that the cryptographic strength of the chosen encryption algorithms is such that the attacker cannot actually decrypt the executable or data. How then does an attacker disrupt a system without being able to decrypt and understand? The following are examples of EED attacks:

- **Replay attacks** - consider an EED platform with encrypted instructions. An attacker can simply re-issue an instruction from the RAM to the CPU. What does such an attack achieve? The application program logic can be disrupted and the resulting behavior exploited by an attacker. Indeed, by observing the results of a multitude of replay attacks, an attacker can catalogue information

about the results of individual replay attacks and use such attacks in tandem for greater disruption. A sample replay attack is depicted in Fig. 2.2.

- **Control-flow attacks** - the sophisticated attacker can elucidate the control-flow structure of a program without decryption. This can be done by simply sniffing the bus, recording the pattern of accesses and extracting the control-flow graph from the list of accesses. To disrupt, an attacker can prematurely transfer control out of a loop, or can transfer control to a distant part of the executable. A sample control-flow attack is depicted in Fig. 2.3.
- **Runtime data attacks** - by examining the pattern of data write-backs to RAM, the attacker can determine the location of the run-time stack and heap even when they are encrypted. By swapping contents in the stack, the attacker can disrupt the flow of execution or parameter passing. Again, the attacker does not need to decrypt to achieve this disruption.
- **Improper processor computations** - the CPU itself may be untrustworthy, since an attacker with considerable resources may simulate the entire CPU and selectively change outputs back to RAM.

Taken together, the above attacks can also be combined with cryptanalytic techniques to uncover cribs for decryption. This suggests that a secure computing platform should be able detect such attacks and prevent disruption.

2.3. Stack Smashing and other Program Flow Attacks

Consider the simple application running in a non-encrypted environment as depicted in Figure 2.4. In this C-style pseudo-code, function `bar` is called with an array of pointers

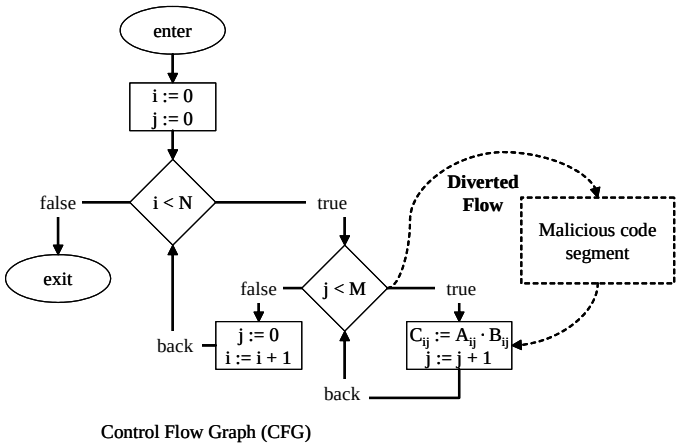

Figure 2.3. A sample Control Flow Graph (CFG) with a diverted flow

to character arrays as its input. These input arrays are copied to a local array using the standard C library `strcpy` call before further processing is applied.

In this code example, since the `strcpy` call does not check to ensure that there is sufficient space in the destination array to fit the contents of the source string, it is possible to write enough data to "overflow" the local array. In this example's process memory layout, the local variables for function `bar` are placed in the stack segment, just on top of run-time variables that are used to ensure proper program flow. The common convention is for the stack to grow backwards in memory, with the top of the stack being placed at lower physical addresses and local arrays growing upwards in memory.

When user input is placed into overflowable buffers, the application is vulnerable to the *stack smashing* attack [**67**]. Widespread exploits including the Code Red and SQLSlammer worms have convincingly demonstrated that it is possible for an attacker to insert arbitrary executable code on the stack. The stack smashing attack works as follows:

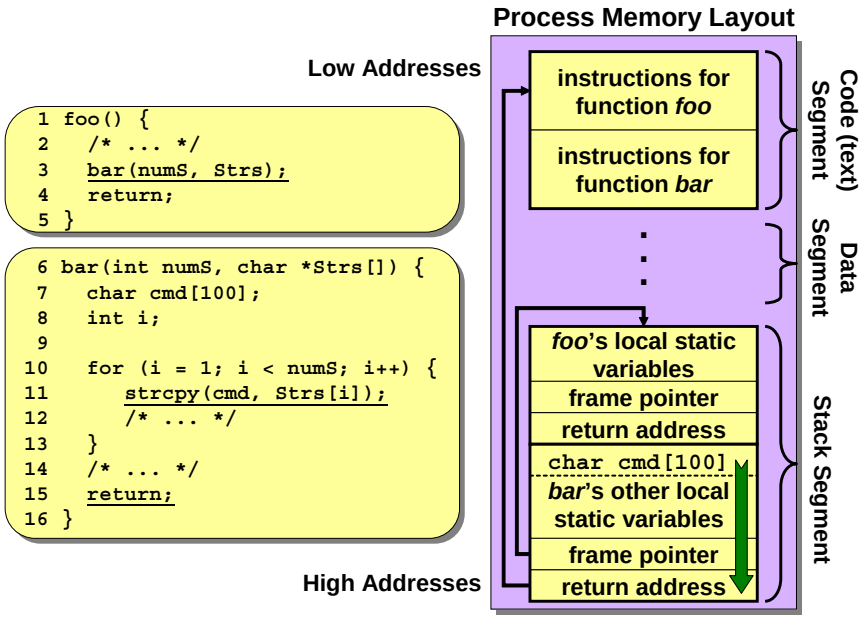

Figure 2.4. Process memory layout of an application with a potentially exploitable buffer overflow

- The attacker first fills the buffer with machine instructions, often called the *shell-code* since typical exploit demonstrations attempt to open a command-shell on the target machine.
- The remainder of the stack frame is filled until the return address is overwritten with a pointer back to the start of the shellcode region.
- When the vulnerable function attempts to return to its calling parent, it uses the return address that has been stored on the stack. Since this value has been overwritten to point to the start of the attacker-inserted code, this is where the program will continue.

The widespread notoriety of stack smashing exploits have led to a considerable amount of focus on their detection and prevention. In the future, it is likely that more complex attacks involving buffer overflows (see [**68**] for a description of *arc injection* and *pointer*

subterfuge) will gain in popularity. Note that any arbitrary program flow modification can be potentially malicious, not just those involving unprotected buffers.

CHAPTER 3

Background and Related Work

While the term *software protection* often refers to the purely software-based defense mechanisms available to an application after all other safeguards have been broken, in practical systems this characterization is not necessarily so precise. For our purposes, we classify hardware-supported secure systems as tools for software protection as long as they include one of the three commonly found elements of a software-based protection scheme [**19**] detailed below. A survey of several software protection techniques appears in [**32**]. A survey of the broader area of software security in the context of Digital Rights Management (DRM) appears in [**16, 92**]. Our approach of embedding binary codes into executables resembles steganography [**48**] in some ways but that the goals and details are quite different.

Watermarking is a technique whereby messages are hidden inside a piece of software in such a way that they can be reliably identified [**18**]. While the oldest type of watermarking is the inclusion of copyright notices into both code and digital media, more recent watermarking approaches have focused on embedding data structures into an application, the existence of which can then be verified at run-time. Venkatesan *et al.* present an interesting variation on watermarking in [**83**], where the watermark is a subprogram that has its control flow graph merged with the original program in a stealthy manner.

The concept behind **tamper-proofing** is that a properly secured application should be able to safely detect at run-time if it has been altered. In [**15**] the authors propose the

concept of *guards*, pieces of executable code that typically perform checksums to guard against tampering. In [**41**], the authors propose a dynamic self-checking technique to improve tamper resistance. The technique consists of a collection of "testers" that test for changes in the executable code as it is running and report modifications. A tester computes a hash of a contiguous section of the code region and compares the computed hash value to the correct value. An incorrect value triggers the response mechanism. They note that performance is invariant until the code size being tested exceeds the size of the L2 cache. A marked deterioration in performance was observed after this occurred. An interesting technique is proposed by Aucsmith in [**8**], in which partitioned code segments are encrypted and are handled in a fashion such that only a single segment is ever decrypted at a time.

These techniques strongly rely on the security of the checksum computation itself. If these checksum computations are discovered by the attacker, they are easily disabled. Moreover, in many system architectures, it is relatively easy to build an automated tool to reveal such checksum-computations. For example, a control-flow graph separates instructions from data even when data is interspersed with instructions; then, checksum computations can be identified by finding code that operates on code (using instructions as data). This problem is acknowledged but not addressed in [**41**].

Proof-Carrying Code (PCC) is a recently proposed solution that has techniques in common with other tamper-proofing approaches. PCC allows a host to verify code from an untrusted source [**64, 62, 63, 5, 9, 11**]. Safety rules, as part of a theorem-proving technique, are used on the host to guarantee proper program behavior. Applications include browser code (applets) [**9**] and even operating systems [**64**]. One advantage of

proof-carrying software is that the programs are self-certifying, independent of encryption or obscurity. The PCC method is effectively a self-checking mechanism and is vulnerable to the same problems that arise with the code checksum methods discussed earlier; in addition they are static methods and do not address changes to the code after instantiation.

The goal of **obfuscation** is to limit code understanding through the deliberate mangling of program structure - a survey of such techniques can be found in [**20**]. Obfuscation techniques range from simple encoding of constants to more complex methods that completely restructure code while maintaining correctness [**21, 20**]. Other authors [**86, 85**] also propose transformations to the code that make it difficult to determine the control flow graph of the program, and show that determining the control flow graph of the transformed code is NP-hard. Similar to tamper-proofing, obfuscation can only make the job of an attacker more difficult, since tools can be built to automatically look for obfuscations, and tracing through an executable in a debugger can reveal vulnerabilities. These and other theoretical limitations are discussed in more detail in [**10**].

Another software technique is to create a system of customizable operating systems wherein much of the security, checksumming and obfuscation are hidden. However, in practice this solution will take longer to find acceptance for several reasons. First, it is expensive to maintain multiple flavors of operating systems. Second, the main vendor of entrenched operating systems fiercely protects code sources – a political act may be needed to require such customizability. Third, the installed base of operating systems is already quite high, which would leave many existing systems still open to attack.

3.1. Other Hardware-based Approaches

Using our definition, there have been several hardware-based software protection approaches. **Tamper-resistant packaging** can coat circuit boards or encase the entire device, such as the iButton developed by [**26**]. **Secure coprocessors** are computational devices that enable execution of encrypted programs. Programs, or parts of the program, can be run in an encrypted form on these devices thus never revealing the code in the untrusted memory and thereby providing a tamper resistant execution environment for that portion of the code. A number of secure coprocessing solutions have been designed and proposed, including systems such as IBM's Citadel [**90**], Dyad [**81, 82, 101, 100**], the Abyss and μAbyss systems [**89, 88, 87**], and the commercially available IBM 4758 which meets the FIPS 140-1 Level 4 validation [**42, 76, 75**]. Distributed secure coprocessing is achieved by distributing a number of secure coprocessors and some have augmented the Kerberos system by integrating secure coprocessing into it [**46**].

Smart cards can also be viewed as type of secure coprocessor; a number of studies have analyzed the use of smart cards for secure applications [**33, 52, 72, 65, 75**]. Sensitive computations and data can be stored in the smart card but they offer no direct I/O to the user. Most smart card applications focus on the secure storage of data although studies have been conducted on using smart cards to secure an operating system [**17**]. As noted in [**15**], smart cards can only be used to protect small fragments of code and data. In addition, as mentioned in [**51, 75**], low-end devices derive their clocks from the host and thus are susceptible to attacks that analyze data dependent timing.

Recent commercial hardware security initiatives have focused primarily on cryptographic acceleration, domain separation, and trusted computing. These initiatives are

intended to protect valuable data against software-based attacks and generally do not provide protection against physical attacks on the platform. MIPS and VIA have added cryptographic acceleration hardware to their architectures. MIPS Technologies' SmartMIPS ASE [**61**] implements specialized processor instructions designed to accelerate software cryptography algorithms, whereas VIA's Padlock Hardware Security Suite [**84**] adds a full AES encryption engine to the processor die. Both extensions seek to eliminate the need for cryptographic coprocessors.

Intel's LaGrande Technology [**43**], ARM's TrustZone Technology [**7**], and MIPS Technologies' SmartMIPS ASE implement secure memory restrictions in order to enforce domain separation. These restrictions segregate memory into secure and normal partitions and prevent the leakage of secure memory contents to foreign processes. Intel and ARM further strengthen their products' domain separation capabilities by adding a processor privilege level. A Security Monitor process is allowed to run at the added privilege level and oversee security-sensitive operations. The Security Monitor resides in protected memory and is not susceptible to observation by user applications or even the Operating System.

Several companies have formed the so-called Trusted Computing Group (TCG) to provide hardware-software solutions for software protection [**80**]. The TCG defines specifications for the Trusted Platform Module, a hardware component that provides digital signature and key management functions, as well as shielded registers for platform attestation. Intel's LaGrande Technology and Microsoft's Next-Generation Secure Computing Base combine the TPM module with processor and chipset enhancements to enable platform attestation, sealed storage, strong process isolation, and secure I/O channels. All

of these approaches require processor or board manufacturers to commit to a particular design, and once committed, are locked into the performance permitted by the design.

3.2. Closely Related Work

In [**54**], researchers at Stanford University proposed an architecture for tamper-resistant software based on an eXecute-Only Memory (XOM) model that allows instructions stored in memory to be executed but not manipulated. A hardware implementation is provided that is not dissimilar to our proposed architecture, with specialized hardware being used to accelerate cryptographic functionality needed to protect data and instructions on a per-process basis. Three key factors differentiate our work from the XOM approach. One distinction is that our architecture requires no changes to the processor itself. Also, our choice of reconfigurable hardware permits a wide range of optimizations that can shape the system security and resultant performance on a per-application basis. Most importantly, we consider a host of new problems arising from attacks on these type of encrypted execution and data platforms.

In [**50**], researchers at UCLA and Microsoft Research propose an intrusion prevention system known as the Secure Program Execution Framework (SPEF). Similar to our proposed work, the SPEF system is used as the basis for compiler transformations that both perform code obfuscation and also embed integrity checks into the original application that are meant to be verified at run-time by custom hardware. The SPEF work in its current form concentrates solely on preventing intruder code from being executed, and consequently neglects similar attacks that would focus mainly on data integrity. Also, the compiler-embedded constraints in the SPEF system require a predefined hardware

platform on which to execute; this limits the scope of any such techniques to the original processor created for such a purpose.

Pande *et al.* [**105**] address the problem of information leakage on the address bus wherein the attacker would be snooping the address values to gain information about the control flow of the program. They provide a hardware obfuscation technique which is based on dynamically randomizing the instruction addresses. This is achieved through a secure hardware coprocessor which randomizes the addresses of the instruction blocks, and rewrites them into new locations. While this scheme provides obfuscation of the instruction addresses, thereby providing a level of protection against IP theft, it does not prevent an attacker from injecting their own instructions to be executed and thereby disrupting the processor and application.

3.3. Buffer Overflow Prevention

Several compiler-based solutions currently exist for buffer overflow vulnerabilities. StackGuard [**24**] is a compiler modification that inserts a unique data value above the return address on the stack. The code is then instrumented such that this value is then checked before returning to the caller function. This check will fail if an overflowing buffer modifies this value. StackShield is a similar compiler extension that complicates the attack by copying the return addresses to a separate stack placed in a different and presumably safer location in memory. While they are effective, it should be noted that these protections can be bypassed in certain situations [**13**].

Designating memory locations as non-executable using special hardware tags is becoming a popular method for deterring buffer overflow attacks. Although available on a

variety of older processors, most recently AMD has released hardware with their NX (No eXecute) technology and Intel has followed suit with a differently named yet functionally equivalent XD (eXecute Disable) bit. Software emulation of non-executable memory is a less secure option for processors that do not include functionality similar to the NX bit. These approaches are being widely adopted for general-purpose processors, however they do not address any type of program flow attack that doesn't involve instructions being written to and later fetched and executed from data memory.

There have been several hardware-based approaches to preventing buffer overflow attacks. The authors in [**23**] utilize their DISE architecture to implement a concept similar to StackShield in hardware. Both the SPEF framework [**50**] and the XOM architecture [**54**] mentioned previously can be effective at disallowing the insertion of untrusted code. However, the latency overhead introduced by encrypting the instruction fetch path can be considerable and may not be acceptable in real-time embedded systems; this problem is currently being examined by the computer architecture community [**73, 98**].

3.4. FPGAs and Security

FPGAs have been used for security-related purposes in the past as hardware accelerators for cryptographic algorithms. Along these lines Dandalis and Prasanna [**27, 69**] have led the way in developing FPGA-based architectures for internet security protocols. Several similar ideas have been proposed in [**79, 49**]. The FPGA manufacturer [**2**] offers commercial IP cores for implementations of the DES, 3DES, and AES cryptographic algorithms. These implementations utilize FPGAs not only for their computational speed but for their programmability; in security applications the ability to modify algorithmic

functionality in the field is crucial. In Chapter 7 we describe a series of optimizations that are effective at improving the performance of various cryptographic ciphers.

In order for an FPGA to effectively secure an embedded or any other type of system, there is a requisite level of confidence needed in the physical security of the FPGA chip itself. In an attempt to raise consumer confidence in FPGA security, [**3**] is currently developing new anti-fuse technologies that would make FPGAs more difficult to reverse-engineer. This added physical security greatly enhances the value of our approach, since as hardware technologies improve to make the "secure component" more secure, the level of trust of the software running on a target embedded system significantly increases.

CHAPTER 4

The SAFE-OPS Software Protection Framework

Our approach, called SAFE-OPS (Software/Architecture Framework for the Efficient Operation of Protected Software), considers the following broad themes. First, designers would like the ability to fine-tune the level of security for a desired level of performance. Second, it may not be practical to store a secret key within the FPGA in small embedded processors; for such systems, a simpler approach is recommended. Third, for performance reasons, it is possible to inhibit tampering without necessarily addressing code understanding. This paper presents an approach whose level of security lies between the two extremes and can be tuned within this range. Furthermore, by exploiting the wide availability and re-programmability of FPGA (Field-Programmable Gate Array) hardware, these techniques will be applicable in the near future (3-5 years) and are compatible with System-on-Chip (SoC) designs that feature processor cores, interconnects and programmable logic.

The proposed method works as follows (see Figure 4.1). The processor is supplemented with an FPGA-based *secure hardware component* that is capable of fast decryption and, more importantly, capable of recognizing and certifying strings of keys hidden in regular unencrypted instructions. To allow a desired level of performance (execution speed), the compiler creates an executable with parts that are encrypted (the "slow" parts) and parts that are unencrypted but are still tamper-proof. Thus, in Figure 4.1, the first part of the executable is encrypted and will be decrypted by the FPGA using a standard private

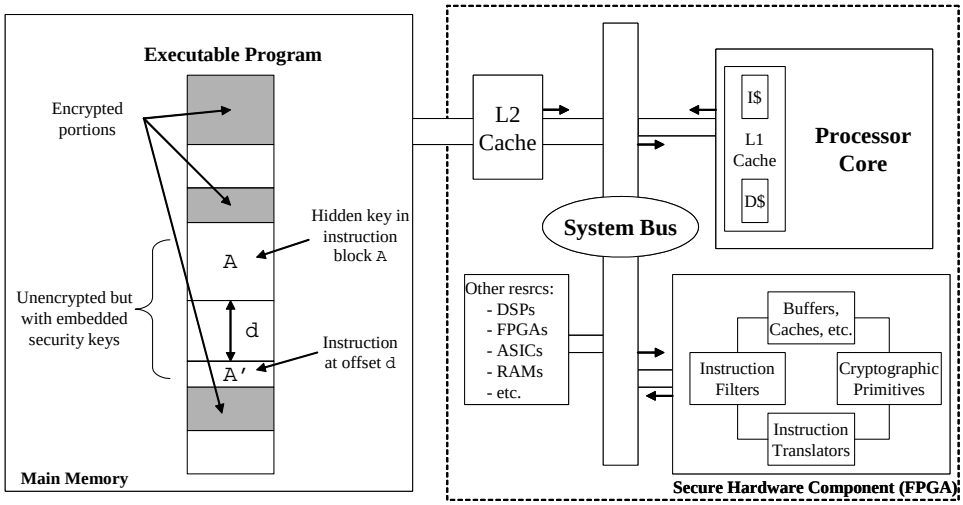

Figure 4.1. Conceptual view of the SAFE-OPS framework

(secret) key technique [**25**]. The second part of the executable shows instruction block A containing a hidden key and, at a distance d from A, an instruction A'. Upon recognizing A, the FPGA will expect $A' = f(A)$ at distance d (where f is computed inside the FPGA); if the executable is tampered with, this match is highly unlikely to occur and the FPGA will halt the processor. The key sequences can be hidden within both instructions and data. The association of A' with A is hidden even from invasive electronic probing. The FPGA waits for the processor to fetch A and A' during program execution before checking the key. The programmability of the FPGA together with the compiler's ability to extract program structure and transform intermediate instructions provide the broad range of parameters to enable security-performance tradeoffs.

The strength of this approach is tunable in several ways. The key length and the mapping space can be increased, but at a computational cost. An optional secret key can be used to make f a cryptographic hash [**12**] for increased security. By only using register codes that are examined by the FPGA, a lower level of security is provided, but

the executable instructions are left compatible with processors that do not contain the FPGA component. The computations performed in the FPGA are very efficient: there are counters, bit-registers and comparators. All of these operations can be performed within a few instruction cycles of a typical CPU. Later on in this work, this assumption is relaxed with an investigation of the performance benefits of the placement of more complex functionality on the FPGA.

At its highest level, the SAFE-OPS approach is best classified as a combination of the tunability of classical compiler-based software protection techniques with the additional security afforded through hardware. Our work is unique in that we utilize a combined hardware/software technique, and that we provide tremendous flexibility to application designers in terms of positioning on the security/performance spectrum. Also, going back to Fig. 1.1, our approach improves upon previous software protection attempts by accelerating their performance without sacrificing any security.

4.1. Architecture Overview

We accomplish our low-overhead software protection through the placement of an FPGA between the highest level of on-chip cache and main memory in a standard processor instruction stream (see Fig. 4.2). In our architecture, the FPGA traps all instruction cache misses, and fetches the appropriate bytes directly from higher-level memory. These instructions would then be translated in some fashion and possibly verified before the FPGA satisfies the cache request. Both the translation and verification operations could be customized

The key features of our software protection architecture can be summarized as follows:

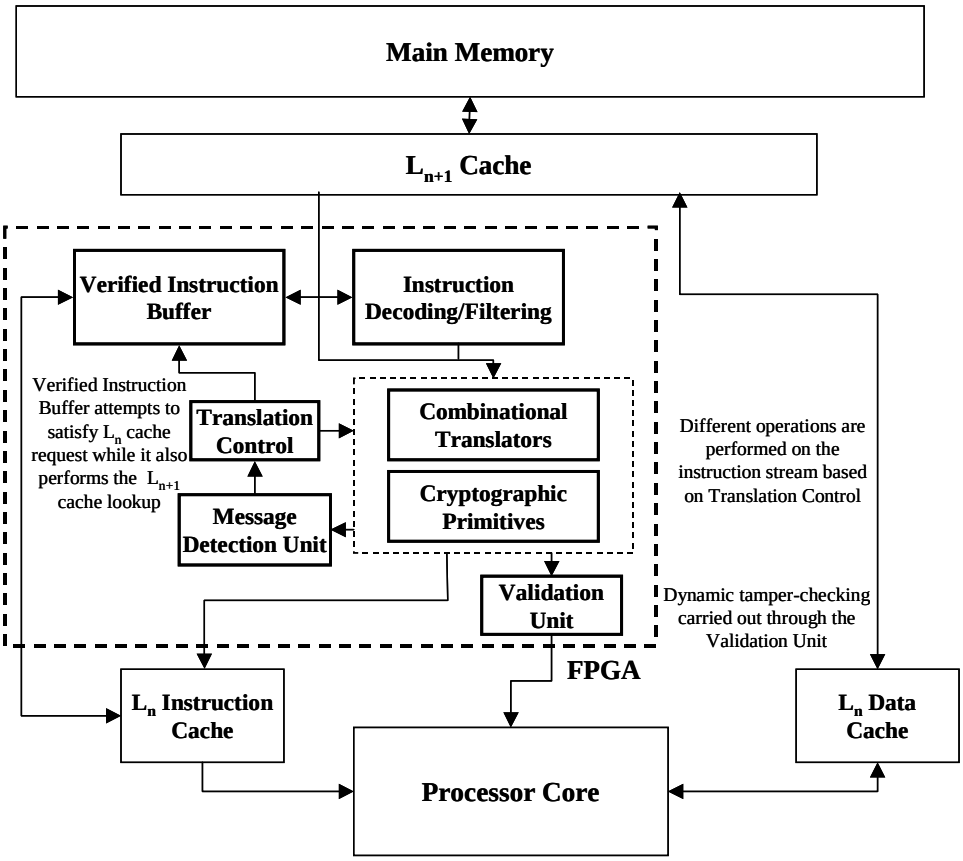

Figure 4.2. FPGA-Based software protection architecture

- **Fast Decryption** - Much work has been done in recent years on FPGA implementations of cryptographic algorithms; a popular target is the Advanced Encryption Standard (AES) candidate finalists [**28**]. These results have shown that current-generation FPGAs are capable of extremely fast decryption. As an example in [**47**] an implementation of the AES cipher attained a throughput far exceeding 10 Gbps when fully pipelined. Consequently, by placing AES hardware on our FPGA we can perform instruction stream decryption without the prohibitive delays inherent in an equivalent software implementation.

- **Instruction Translation** - As a computationally less complex alternative to the full decryption described above, we can instantiate combinational logic on the FPGA to perform a binary translation of specific instruction fields. For example, a simple lookup table can map opcodes such that all addition instructions become subtractions and vice versa. As the mapping would be completely managed by the application developer, this technique would provide an effective yet extremely low-cost level of obfuscation.
- **Dynamic Validation** - In order to effectively prevent tampering, we can program the FPGA with a set of "rules" that describe correct program functionality. For a code segment that is fully encrypted, these rules can be as simple as requiring that each instruction decode properly. An example of a more complex rule would be one based on a predetermined instruction-based watermark. These software integrity checks can be performed either at each instruction or at a predefined interval - in either case given a detection of a failure condition the FPGA would require special attention from the system.

4.2. Compiler Support

As mentioned earlier, every feature that is implemented in our architecture requires extensive compiler support to both enhance the application binary with the desired code transformations and to generate a hardware configuration for an FPGA that incorporates the predetermined components. Figure 4.3 illustrates how some of the standard stages in a compiler can be enhanced with software protection functionality. Many of the standard stages in a compiler can be modified to include software protection functionality. As an

example, consider the data flow analysis module. In a standard performance-oriented compiler, the goal of this stage is to break the program flow into basic blocks and to order those blocks in a manner that increases instruction locality. However, it is possible to reorganize the code sequences in such a fashion that sacrifices some performance in exchange for an increased level of obfuscation. This is an example of a transformation that requires no run-time support; for others (such as those that use encryption), the compiler must also generate all the information needed to configure an FPGA that can reverse the transformations and verify the output. Our work currently focuses on compiler back-end transformations (i.e. data flow analysis, register allocation, code generation), although there are possiblities in the front-end to examine in the future as well.

4.3. Example Implementations

Up to this point, in detailing the different features available in our software protection architecture, we have maintained a fairly high-level view. In the following section, we delve into the specifics of two examples that illustrate the potential of our combined compiler/FPGA technique.

4.3.1. Tamper-Resistant Register Encoding

Consider the code segment as depicted on the left in Figure 4.4 and focus on the instructions as they are fetched from memory to be executed by the processor (the *instruction stream*). Initially, for illustration, the complexity introduced by loops is not considered. In most instruction set architectures, the set of instructions comprising a sequentially-executed code segment will contain instructions that use registers. In this example, the

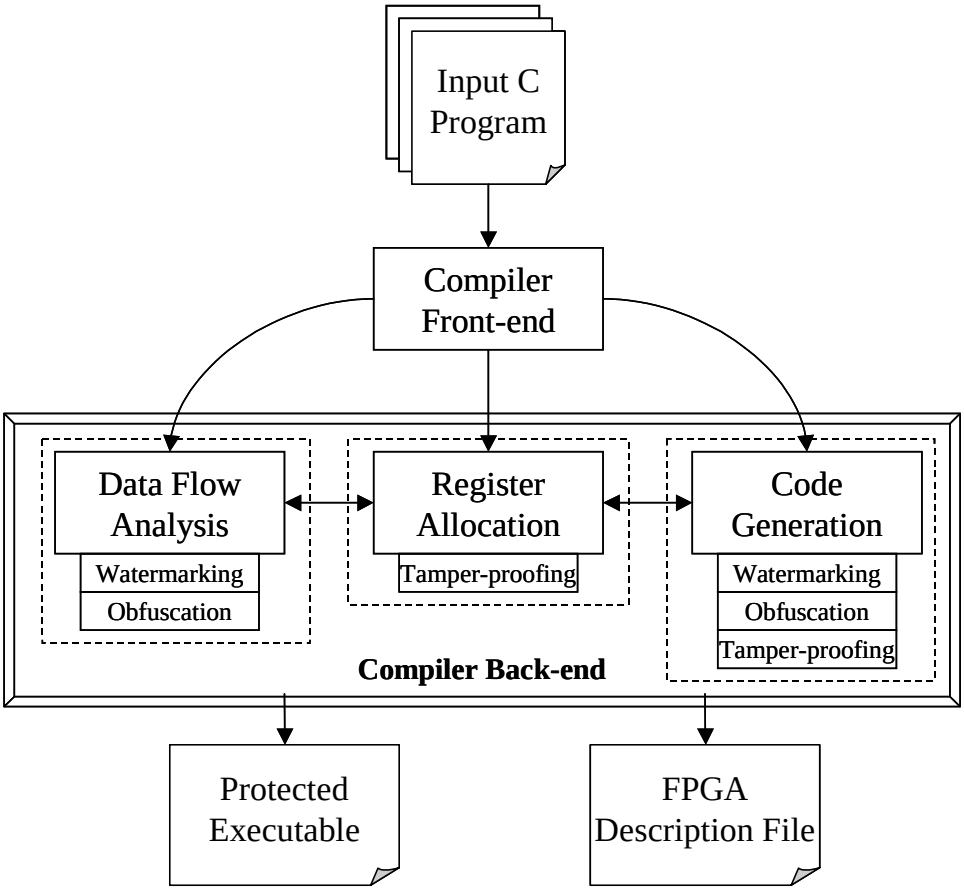

Figure 4.3. HW/SW Compilation Framework

decoding unit in the FPGA will extract one register in each register-based instruction - we can refer to the sequence of registers so used in the instruction stream as the *register stream*. In addition, the FPGA also extracts the opcode stream. In the example, part of the register stream shown is: R_1, R_2, R_2, R_1, R_1, R_2. After being extracted, the register stream is then given a binary encoding - this example shows how R_1 encodes **0** and R_2 encodes **1** and therefore the particular sequence of registers corresponds to the code **0 1 1 0 0 1**. The key observation is that this register stream is determined by the register allocation module of the compiler.

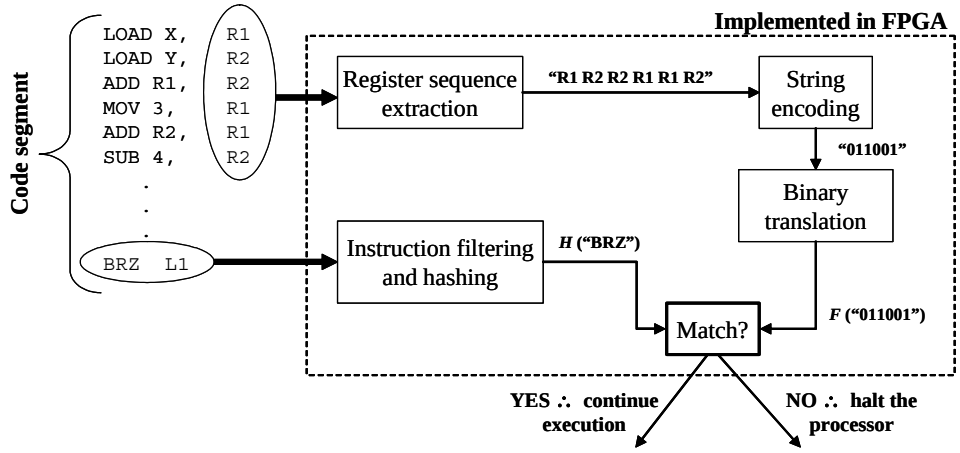

Figure 4.4. Tamper-Resistant Register Encoding

A binary translation component then feeds this string through some combinational logic to generate a key. This function can be as simple as just flipping the bits, although more complex transformations are possible. The key is then compared against a function of the opcode stream. In this example, an instruction filter module picks out a branch instruction following the register sequence (at distance d, as in Figure 4.1) and then compares (the function f in Figure 4.1) the key to a hash of the instruction bits. If there is a match, the code segment is considered valid, otherwise the FPGA warns the system that a violation condition has been detected. This concept is similar to those proposed in [**50**].

The example illustrates the key ideas:

- The compiler performs instruction filtering to decide which instructions in the opcode stream will be used for comparisons.
- The compiler uses the flexibility of register allocation to bury a key sequence in the register stream.

- Upon execution, the instruction stream is piped through the secure FPGA component.
- The FPGA then extracts both the filtered opcodes and the register sequences for comparisons.
- If a violation is detected, the FPGA halts the processor.

How does such an approach work? As register-allocation is performed by the compiler, there is considerable freedom in selecting registers to allow for any key to be passed to the FPGA (the registers need not be used in contiguous instructions since it is only the sequence that matters). Also, the compiler determines which mechanism will be used to filter out the instructions that will be used for comparisons. If the code has been tampered with, there is a very high probability that the register sequence will be destroyed or that the opcode filtering will pick out a different instruction. As an example, if the filtering mechanism picks out the sixth opcode following the end of a register sequence of length k, any insertion or deletion of opcodes in that set of instructions would result in a failure.

The example above shows how compiler-driven register keys can be used for efficiently ensuring code integrity, authorization and obfuscation. By including these keys in compiler-driven data allocation, the fourth goal of data integrity can also be included.

It is important to note that this type of tamper-proofing would not normally be feasible if implemented entirely in software, since the checking computation could be easily identified and avoided entirely. The register-sequence can be used to encode several different items. For example, an authorization code can be encoded, after which other codes may be passed to the secure-component. This technique can also be used to achieve code obfuscation by using a secret register-to-register mapping in the FPGA. Thus, if

the FPGA sees the sequence (R_1, R_2, R_1), this can be interpreted by the FPGA as an intention to actually use R_3. In this manner, the actual programmer intentions can be hidden through using a mapping customized to a particular processor. The complexity of the transformation can range from simple (no translation) to complex (private-key based translation). Such flexibility brings with it tradeoffs in hardware in terms of speed and cost.

This technique can also be used to achieve code obfuscation by using a secret register-to-register mapping in the FPGA. Thus, if the FPGA sees the sequence (r_1, r_2, r_1), this can be interpreted by the FPGA as an intention to actually use r_3. In this manner, the actual programmer intentions can be concealed by using a mapping customized to a particular processor.

When examining a block of code to be encrypted using this scheme, often the compiler will lack a sufficient number of register-based instructions to encode the desired custom key. In this case, "place-holder" instructions which contain the desired sequence values but which otherwise do not affect processor state, can be inserted by the compiler. Figure 4.5 shows a sample basic block where the desired register sequence values are obtained using a combination of register allocation and instruction insertion. Introducing instructions adds a performance penalty which is examined in the following chapter.

4.3.2. Selective Basic Block Encryption

Selective encryption is a useful technique in situations where certain code segments have high security requirements and a full-blown cryptographic solution is too expensive from a performance perspective. The example code segment in Fig. 4.6 shows how the compiler

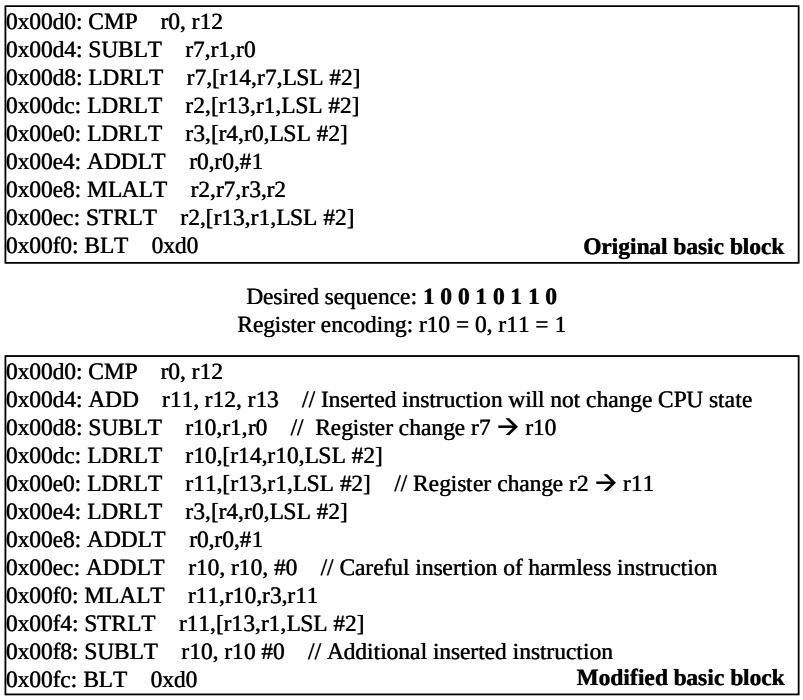

Figure 4.5. Obtaining a desired register sequence value

could insert message instructions to signify the start of an encrypted basic block (the compiler would also encrypt the block itself). As this message is decoded by the FPGA, an internal state would be set that directs future fetched instructions to be fed into the fast decryption unit. These control signals could be used to distinguish between different ciphers or key choices. The freshly-decrypted plaintext instructions would then be validated before being returned to the L_n cache of Fig. 4.2. The encryption mode could then be turned off or modified with the decoding of another message instruction.

Although properly encrypting a code segment makes it unreadable to an attacker who does not have access to the key, using cryptography by itself does not necessarily protect against tampering. A simple way of verifying that instructions have not been tampered with is to check if they decode properly based on the original instruction set

architecture specification. However, this approach does not provide a general solution, as the overwhelming portion of binary permutations are usually reserved for the instruction set of most processors. This increases the likelihood that a tampered ciphertext instruction would also decode properly. A common approach is the use of one-way hash functions (the so-called "message digest" functions), but in our case, it would be prohibitively slow to calculate the hash of every encrypted basic block in even medium-sized applications. A more simple approach would be to recognize patterns of instructions in the code segment that make sense in terms of the register access patterns. Specific care must also be taken to ensure the integrity of the message instructions themselves. This can be implemented through a combination of the register encoding techniques discussed previously and other dynamic code checking methods.

This example describes an approach that would be inherently slow in a purely-software implementation. Consequently, using the FPGA for decryption allows the application designer the flexibility to either improve the overall performance or increase the level of security by encrypting a greater number of code subsections while still meeting the original performance constraints.

4.4. Advantages of our Approach

Based on these two previous examples, we can summarize the advantages that our approach contains over current software protection methodologies as follows:

(1) Our approach simultaneously addresses multiple attacks on software integrity by limiting both code understanding and code tampering.

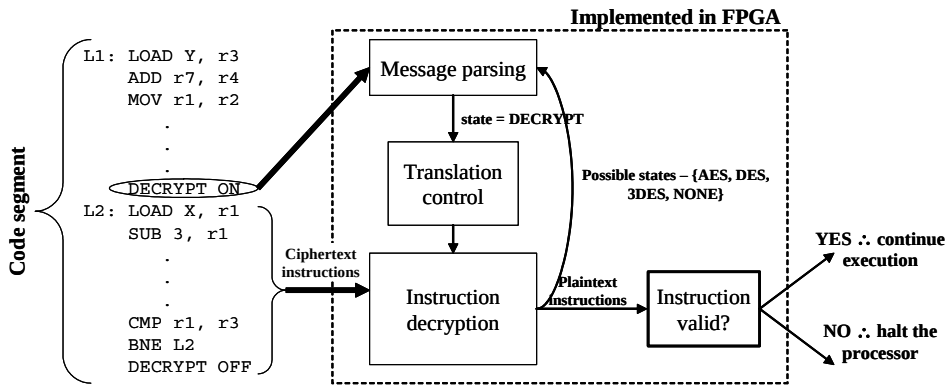

Figure 4.6. Selective Basic Block Encryption

(2) The compiler's knowledge of program structure, coupled with the programmability of the FPGA, provides the application developer with an extreme amount of flexibility in terms of the security of the system. The techniques available to any given application range from simple obfuscation that provides limited protection with a minimal impact on performance to a full cryptographic solution that provides the highest level of security with a more pronounced impact on performance.

(3) Our approach provides the ability to combine both hardware specific techniques with hardware-optimized implementations of several of the software-based methods proposed recently [**15, 41**].

(4) The selection of the FPGA as our secure component minimizes additional hardware design. Moreover, the choice of a combined processor/FPGA architecture enables our system to be immediately applicable to current SoC designs with processors and reconfigurable logic on-chip, such as the Xilinx Virtex-II Pro architecture [**94**].

4.5. Security Analysis of FPGA Hardware

Thus far, we have been working off the assumption that as the software protection mechanisms are situated directly in hardware (FPGA), these dynamic checks are inherently more secure than those validated through purely-software mechanisms. While this may very well be the case, some recent work has suggested that the use of FPGA hardware in embedded systems invites a range of attacks. For example, physical attacks that take advantage of the fact that not all data is lost in a Static RAM (SRAM) memory cell when its power is turned off have been shown to be very successful at retrieving information from memory devices that use SRAM technology [**52**]. It is likely that the techniques developed to facilitate these types of attacks would also be useful in extracting information from SRAM FPGAs. Another point of weakness is the FPGA bitstream. In our system, this file would need to be stored on disk just as the application itself. This potentially exposes some of our techniques to a reverse engineering attack.

Before we lose hope in the prospect of an FPGA as a secure hardware component, several clarifications need to be made. First, many of the proposed attacks on FPGAs assume that the physical security of the entire system has already been breached. No system, regardless of the underlying hardware or software security, is safe assuming that the physical security has been breached. Consequently, under such assumptions, the choice of FPGA provokes no further risks in terms of physical security. Second, reverse engineering a hardware description is a considerably more difficult task than reverse-engineering an equivalent software description, due to the lack of readily available tools and the increased complexity of hardware designs. This task can be made even more difficult through the encrypting of the bitstream itself. These reasons, combined with

the fact that newer FPGAs are being built with highly secure antifuse and flash-based technologies, have motivated the argument made by some that FPGAs are more difficult to reverse-engineer than an equivalently functional ASIC [**2**].

We have up to this point in this thesis demonstrated the usefulness of our architecture by providing specific examples of how our approach can be used in a software protection scheme. What remains to be seen, however, is to the extent to which the insertion of the FPGA in the instruction memory hierarchy effects system security and performance. We address this question in the following chapter.

CHAPTER 5

Framework Evaluation

Since the majority of the techniques we leverage operate on a single program basic block at a time, it makes sense to analyze the effect of the FPGA on instruction memory hierarchy performance at that level of granularity. We begin by considering the replacement penalty of a single block of an L1 cache directly from a pipelined main memory. With a block size of n_{bytes} and a memory width of W_{mem} bytes, we can write the penalty as:

$$t_{miss} = t_{nseq} + \left(\left\lceil \frac{n_{bytes}}{W_{mem}} \right\rceil - 1 \right) \cdot t_{seq} \quad , \tag{5.1}$$

where t_{nseq} is the access latency for the first bytes from memory and t_{seq} is the pipelined latency ($t_{nseq} \geq t_{seq}$). Now, considering a basic block B_i of length n^i_{instrs} instructions in isolation from its surrounding program, we can estimate the number of instruction cache misses (assuming W_{instr} bytes per instruction) as:

$$n^i_{miss} = \left\lceil \frac{W_{instr} \cdot n^i_{instrs}}{n_{bytes}} \right\rceil \quad , \tag{5.2}$$

as each instruction in B_i would be fetched sequentially. Consequently the total instruction cache miss delay can be approximated by the product of n^i_{miss} and t_{miss}.

What effect would our software protection architecture have on performance? We identify two dominant factors: the insertion of new instructions into the executable and

the placement of the FPGA into the instruction fetch stream. Looking now to modified basic block $B_j = f(B_i)$, we can write all changes in the size of the basic block quite simply as:

$$n^j_{instrs} = n^i_{instrs} + \Delta^i_{instrs} \quad , \tag{5.3}$$

and consequently the number of instruction cache misses would increase by a function of Δ^i_{instrs}:

$$n^j_{miss} = \left\lceil \frac{W_{instr} \cdot n^j_{instrs}}{n_{bytes}} \right\rceil \approx n^i_{miss} + \left\lceil \frac{W_{instr} \cdot \Delta^i_{instrs}}{n_{bytes}} \right\rceil \tag{5.4}$$

The majority of the operations performed in the FPGA can be modeled as an increase in the instruction fetch latency. Assuming a pipelined implementation of whatever translation and validation is performed for a given configuration, we can estimate the delay τ_{miss} for an FPGA that decodes n_{bytes} bytes in bursts of W_{fpga} at a time as:

$$\tau_{miss} = \tau_{nseq} + \left(\left\lceil \frac{n_{bytes}}{W_{fpga}} \right\rceil - 1 \right) \cdot \tau_{seq} \tag{5.5}$$

Therefore, the total slowdown in the L1 instruction cache miss penalty is just a simple function of the inserted instructions and the additional FPGA latency:

$$\begin{aligned} \%_{slowdown} &= \frac{n^j_{miss} \cdot \tau_{miss}}{n^i_{miss} \cdot t_{miss}} \\ &\approx \frac{\left(n^i_{miss} + \left\lceil \frac{W_{instr} \cdot \Delta^i_{instrs}}{n_{bytes}} \right\rceil \right) \cdot \left(\tau_{nseq} + \left(\left\lceil \frac{n_{bytes}}{W_{fpga}} \right\rceil - 1 \right) \cdot \tau_{seq} \right)}{n^i_{miss} \cdot \left(t_{nseq} + \left(\left\lceil \frac{n_{bytes}}{W_{mem}} \right\rceil - 1 \right) \cdot t_{seq} \right)} \end{aligned} \tag{5.6}$$

What effect would our software protection architecture have on performance? Based on these results, we identify two dominant factors: the occasional insertion of new instructions into the executable and the placement of the FPGA into the instruction fetch stream. For the first factor, the inserted instructions will only add a small number of cache misses for the fetching of the modified basic block, since for most cases the number of inserted bytes will be considerably smaller than the size of the cache block itself. For the second factor, we note that the majority of the operations performed in the FPGA can be modeled as an increase in the instruction fetch latency. Assuming a pipelined implementation of whatever translation and validation is performed for a given configuration, we can estimate the delay for our FPGA as that of a similar bandwidth memory device, with a single nonsequential access latency followed by a number of sequential accesses. This overall slowdown will be a function of the following terms: (1) the number of inserted instructions Δ^{i}_{instrs}, (2) the ratio of the nonsequential and pipelined access times of the FPGA and main memory $\frac{\tau_{nseq}}{t_{nseq}}$ and $\frac{\tau_{seq}}{t_{seq}}$, and (3) the ratio of the byte widths of the FPGA and main memory $\frac{W_{fpga}}{W_{mem}}$. In the remainder of this section, we explain our experimental approach and then provide quantitative data that demonstrates how these terms are affected by the security requirements of an application.

5.1. Initial Experiments

Figure 5.1 shows the original SAFE-OPS experimental framework. Using a modified version of the `gcc` compiler targeting the ARM instruction set, register encoding schemes were implemented within the data-flow analysis, register allocation, and code generation

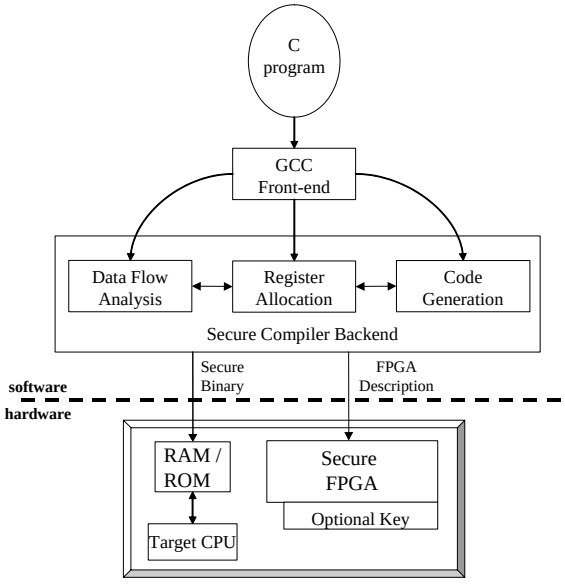

Figure 5.1. SAFE-OPS experimental framework

phases of the `gcc` back-end. The compiler's output is (1) – the encrypted application binary and (2) – a description file for the secure FPGA component.

A custom hardware/software cosimulator was developed in order to obtain performance results for the following experiments. As can be seen in Figure 5.2, the SAFE-OPS simulation infrastructure leverages a commercial software simulator and a commercial HDL simulator, which are connected together by a socket interface. The ARMulator simulator from [**6**] was utilized for modeling the software component of the system. The ARMulator is a customizable instruction set simulator which can accurately model all of the various ARM integer cores alongside memory and OS support. The ARMulator can be customized in two ways: first, by changing system configuration values (clock rate, cache size and associativity, etc.), and second, through pre-defined modules known as the ARMulator API, that can be extended to model additional hardware or software for prototyping purposes.

For simulating the FPGA, the Modelsim simulator from [**60**] was selected. Similar to the ARMulator API, the Modelsim simulator contains a Foreign Language Interface (FLI), which allows for the simulation of C code concurrent with the VHDL or Verilog simulation. A socket interface was set up between the two simulators by using the ARMulator API and the Modelsim FLI. In the general case, when the ARMulator senses a read miss in the instruction cache, the interface code sends a signal over the shared socket to the interface written in the FLI. This signal informs the HDL simulator that an instruction read request has been made, and the appropriate FPGA code is then triggered for execution. When the FPGA finishes loading the instruction from memory and potentially decoding it, a signal is sent back through the socket to the ARMulator containing both the desired instruction along with data signifying how many execution cycles were required. Since the ARM would in general halt execution to handle the event of an instruction cache miss, the total system run-time would be equal to the sum of the total number of cycles required by the ARM for its normal execution and the total number of cycles required by the FPGA. Since the cosimulator handles the separate tasks of hardware and software simulation at a cycle-accurate level, the entire system simulation is also cycle-accurate.

5.1.1. Benchmarks

In order to test the effectiveness of the approach, a diverse set of benchmarks from a variety of embedded benchmark suites were selected. Chosen from the MediaBench [**53**] suite are two different voice compression programs: *adpcm* – which implements Adaptive Differential Pulse Code Modulation decompression and compression algorithms, and *g721* – which implements the more mathematically complex CCITT (International Telegraph

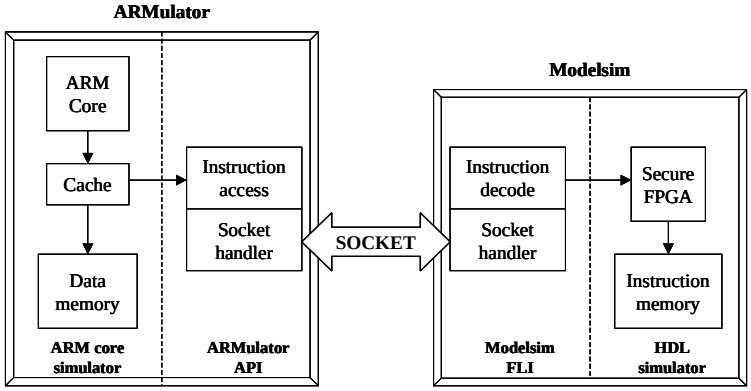

Figure 5.2. HW/SW cosimulator structure

and Telephone Consultative Committee) standard. From the ARM Applications Library that is included with the ARM Developer Suite *arm_fir* – which implements a Finite Impulse Response (FIR) filter. From the MiBench [**36**] embedded benchmark suite three applications were picked: *susan* – an image recognition package that was developed for recognizing corners and edges in Magnetic Resonance Images (MRI) of the brain, *dijkstra* – an implementation of Dijkstra's famous algorithm for calculating shortest paths between nodes, customized versions of which can be found in network devices like switches and routers, and *fft* – which performs a Fast Fourier Transform (FFT) on an array of data. These benchmarks are representative of the tasks that would be required of embedded processors used in multimedia and/or networking systems. More details of the selected benchmarks can be found in Table 5.1.

For the initial experiments, the simulator was configured to model an ARM9TDMI core which contains sixteen, 32-bit general purpose registers. The ARM core is configured to operate at 200MHz and is combined with separate 8KB instruction and data caches.

Benchmark	Source	Code Size	# Basic Blocks	Instr Count (mil)
adpcm	MediaBench	8.0 KB	26	1.23
*g*721	MediaBench	37.9 KB	84	8.67
arm_fir	ARM AppsLib	44.0 KB	34	.301
susan	MiBench	66.4 KB	119	2.22
dijkstra	MiBench	42.5 KB	32	7.77
fft	MiBench	69.2 KB	44	4.27

Table 5.1. Benchmarks used for experimentation

Setting the memory bus clock rate to be 66.7MHz, the cache miss latency before considering the FPGA access time is 150ns ($\sim$30 CPU cycles). This configuration is similar to that of the ARM920T processor.

The default FPGA model runs at a clock speed identical to that of the ARM core and requires an extra 3-5 cycles to process and decode an instruction memory access. However, the clock speed of an FPGA is in general determined by the critical path of its implemented logic. As will be demonstrated in this section, when considering architectural improvements to the FPGA decoder implementation, the potential effect on clock speed will also need to be considered.

5.1.2. Register Sequence Encoding Evaluation

The performance and resultant security of the SAFE-OPS approach was first explored by using this customized HW/SW cosimulator to analyze two main metrics of the register sequence encoding approach: (1) the desired length of the register sequence; and (2) the selection criteria for inserting a sequence inside a suitable basic block. The results of these experiments is presented in the following section.

For the six selected benchmarks the effect of increasing the encoded register sequence length on the overall system performance was simulated in the case when approximately

25% of the eligible basic blocks of each benchmark are encoded using a random-selection algorithm. The results, as shown in Figure 5.3, are normalized to the performance of the un-encrypted case. As can be seen in the figure, these initial results demonstrate the potential security/performance tradeoffs inherent in the SAFE-OPS approach. Overall, for most of the benchmarks the performance is within 80% of the base case (no encryption) when considering sequence lengths up to 16. However, when considering the most extreme case (when the sequence length is 32), two of the benchmarks suffer a performance penalty of over 25%. This decreased performance is due to the fact that (1) the inserted instructions require extra cycles for their execution and that (2) the increased code size can lead to more instruction cache misses. This explains why the two largest benchmarks, *susan* and *fft*, performed the best of the set. It is also interesting to note that two other benchmarks *arm_fir* and *dijkstra* of similar sizes do not follow the same performance trends; *arm_fir* performs almost as well as *susan* for each configuration while *dijkstra* does not. This can be explained by the fact that *dijkstra* is a much longer running benchmark, with several basic blocks with extremely high iteration counts. It is very likely that the random-selection algorithm encoded one or more or these high frequency blocks. As will be explained in the following section it is possible to select the basic blocks in a more intelligent fashion that takes these types of loop structures into account.

In Figure 5.4 the case is now considered where the register sequence length is kept at a constant value of 8, and the effect of the aggressiveness of the random basic block selection technique is examined. As the upper limit on the number of encoded basic blocks is increased, it can be seen that there is a limit to the performance of the resulting code. For the majority of the benchmarks, the performance in even the most secure case

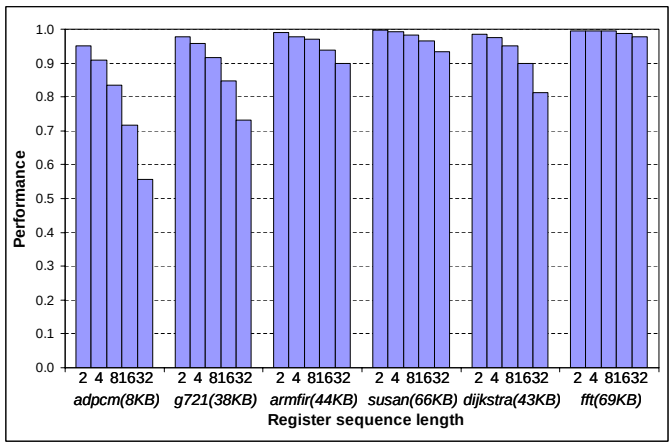

Figure 5.3. Performance as a function of the register sequence length, normalized to the performance of the unencoded benchmarks. When the percentage of selected basic blocks is held constant (25%), increasing the level of security (sequence length) has a negative impact on performance. For most benchmarks, performance remained within 80% of the base case until considering the longest sequence lengths.

is within 75% of the unencrypted case. However, for the two smallest benchmarks (*adpcm* and *g721*), encrypting more than 50% of the eligible basic blocks can have a drastic effect on performance. These results show that if it is possible for the SAFE-OPS compiler to select the right basic blocks to be encoded with an appropriate sequence length value, one would be able to keep the performance at an acceptable level while still increasing security. These results motivate the development of compiler algorithms that utilize profiling and in-depth application analysis techniques in order to make better choices for selecting basic blocks. Two such approaches are discussed later in this chapter.

5.2. Additional Experiments

In order to provide more insight into the performance profile of these applications, we conducted additional experiments, for which incorporated a behavioral model of our

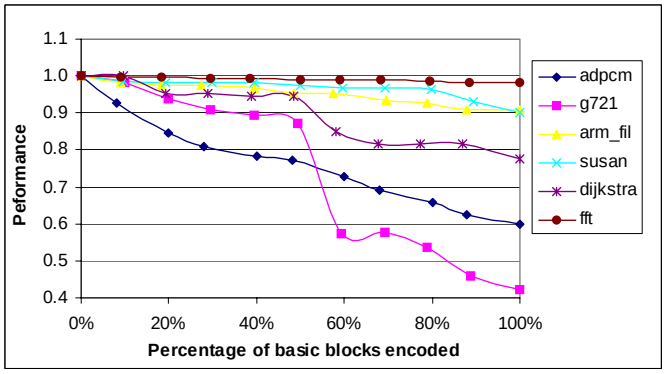

Figure 5.4. Performance as a function of the rate of encoding basic blocks, normalized to the performance of the unencoded benchmarks. While for all of the benchmarks the performance penalty when encoding less than half of the basic blocks is less than 25%, there is a potential for severe performance degradation when the overwhelming majority of the basic blocks are selected. These results motivate the development of basic block selection heuristics that take expected performance into account.

software protection FPGA hardware into the SimpleScalar/ARM tool set [**14**], a series of architectural simulators for the ARM ISA. We examined the performance of our techniques for a memory hierarchy that contains separate 16 KB 32-way associative instruction and data caches, each with 32-byte lines. With no secondary level of cache, our non-sequential memory access latency is 100 cycles and our sequential (pipelined) latency is 2 cycles. This configuration corresponds to the values of $n_{bytes} = 32$, $t_{nseq} = 100$, and $t_{seq} = 2$ from the equations in the previous section. Since our simulation infrastructure was no longer tied to a closed-source compiler technology, we were able to insert our protective techniques directly into the back-end of a modified version of the `gcc` compiler targeting the ARM instruction set.

To evaluate our approach, we adapted six benchmarks from two different embedded benchmark suites. From the MediaBench [**53**] suite we selected two different voice compression programs: *adpcm* – which implements Adaptive Differential Pulse Code Modulation decompression and compression algorithms, and *g721* – which implements the more mathematically complex CCITT (International Telegraph and Telephone Consultative Committee) standard. We also selected from MediaBench the benchmark *pegwit*, a program that implements elliptic curve algorithms for public-key encryption. From the MiBench [**36**] embedded benchmark suite we selected three applications: *cjpeg* – a utility for converting images to JPEG format through lossy image compression , *djpeg* – which reverses the JPEG compression, and *dijkstra* – an implementation of the famous Dijkstra's algorithm for calculating shortest paths between nodes, customized versions of which can be found in network devices like switches and routers.

It should be noted that, as in our original experiments, both our simulation target and selected workload characterize a processor that could be found in a typical embedded system. This choice was motivated by the fact that commercial chips have relatively recently been developed that incorporate both embedded processors and FPGAs onto a single die (ex. Xilinx Virtex-II Pro Platform FPGAs [**94**]). Consequently, even though we see our techniques as one day being useful to a wide range of systems, the tradeoffs inherent in our approach can be best demonstrated through experiments targeting embedded systems, as these results are directly applicable to current technology.

5.2.1. Detailed Performance Breakdown

Using this simulation platform, we first explored the effects of our approach on performance and resultant security with an implementation of the tamper-resistant register encoding example from the previous section. In this configuration, the compiler manipulates sequences of register-based instructions to embed codes into the executable which are verified at run-time by the FPGA. As the operations required are relatively simple, for our experiments we assumed that the operations performed by the FPGA to decode individual instructions requires 1 clock cycle (i.e. $\tau_{sec} = 1$), and that verifying an individual basic block by comparing a function of the register sequence with a function of the filtered instruction requires 3 clock cycles (i.e. $\tau_{sec} = 3$).

As mentioned previously, it is often the case that the compiler will not have enough register-based instructions in a given basic block with which to encode a relatively long sequence string. Consequently, in these cases the compiler must insert Δ^{i}_{instrs} instructions into the basic block which encode a portion of the desired sequence but which otherwise do not affect processor state. However, as these "dummy" instructions must also be fetched from the instruction cache and loaded in the processor pipeline they can cumulatively have a significant detrimental impact on performance.

While quantifying the security of any system is not a simple task, we can estimate the overall coverage of an approach independent of the instantiated tamper-checking computations. For our register encoding approach, we can measure this aggressiveness as a function of both the desired register sequence length and the percentage of basic blocks that are protected. Figure 5.5 considers the performance of our system when the sequence length is kept at a constant value 8 and the percentage of encoded basic blocks is varied

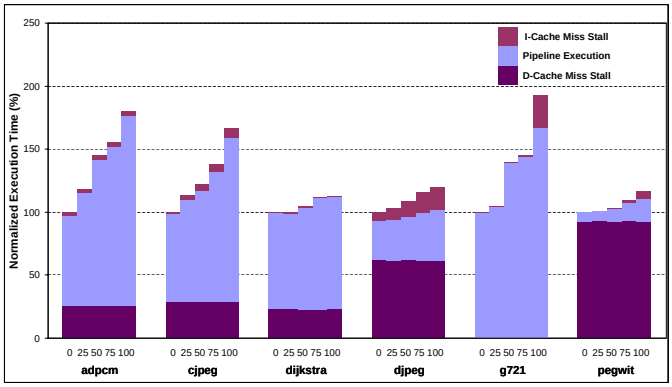

Figure 5.5. Performance breakdown of the tamper-resistant register encoding scheme as a function of the basic block select rate. These results show that for the most secure case (rate = 100) the average performance degredation is approximately 48%.

from 0 - 100%. For each experiment the results are normalized to the unsecured case and are partitioned into three subsets: (1) the stall cycles spent in handling data cache misses, (2) the "busy" cycles where the pipeline is actively executing instructions, and (3) the stall cycles spent in handling instruction cache misses.

Several general points can be made about the results in Figure 5.5. First, it is obvious that the selected embedded benchmarks do not stress even our relatively-small L1 instruction cache, as on average these miss cycles account for less than 3% of the total base case run-time. This is a common trait among embedded applications, as they can often be characterized as a series of deeply-nested loops that iterate over large data sets (such as a frame of video). This high inherent level of instruction locality means that, although the inserted register sequence instructions do affect the cycles spent in handling instruction cache misses, there is a relatively significant negative impact on the non-stall pipeline cycles. The average slowdown for this scheme is approximately 48% in the case where all the basic blocks are selected, but then drops to 20% if we only select half the

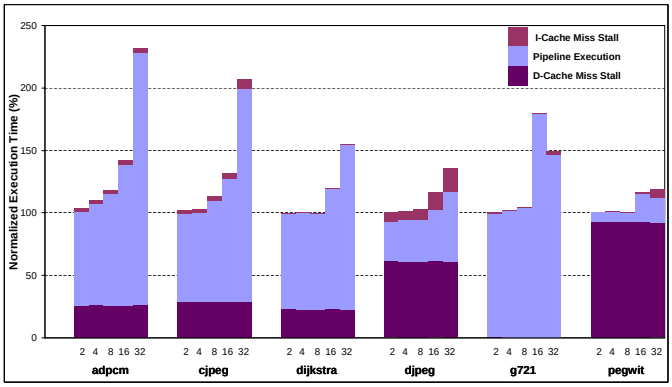

Figure 5.6. Performance breakdown of the tamper-resistant register encoding scheme as a function of the register sequence length. These results show that for the most secure case (length = 32) the average performance degredation is approximately 66%.

blocks. These results clearly demonstrate the tradeoff between security and performance that can be managed in our approach.

In Figure 5.6, we now consider the case where the basic block selection rate is kept constant at 25 and the effect of the desired register sequence length is examined. Although for most of the benchmark/sequence combinations the performance impact is below 50%, there are cases where lengthening the sequences can lead to huge increases in execution time, as the basic blocks are too short to supply the needed number of register-based instructions. This can be seen in the results for the most secure configuration (sequence length value of 32), where the average performance penalty is approximately 66%.

In our next set of experiments, we investigated the performance impact of the selective basic block encryption scheme. FPGA implementations of symmetric block often strive to optimize for either throughput or area. Recent implementations of the Advanced Encryption Standard (AES) have reached a high throughput by unrolling and pipelining the algorithmic specification [**103**]. The resultant pipeline is usually quite deep. Consequently

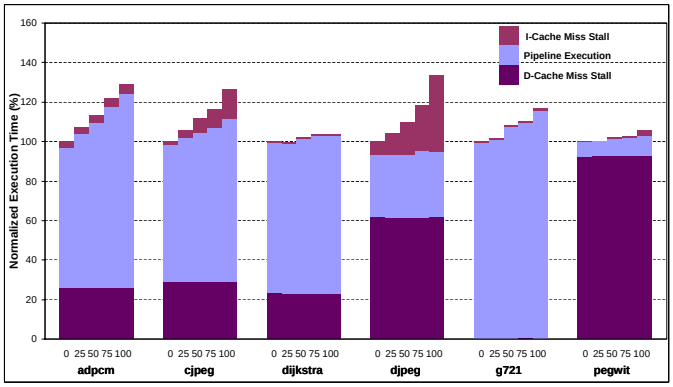

Figure 5.7. Performance breakdown of the selective basic block encryption scheme as a function of the block select rate. These results show that for the most secure case (rate = 100) the average performance degredation is approximately 20%.

the initial access latencies are over 40 cycles, while the pipelined transformations can be executed in a single cycle. Assuming an AES implementation on our secure FPGA, we can make the intelligent assumption of $\tau_{nseq} = 50$ and $\tau_{seq} = 1$.

Figure 5.7 shows the performance breakdown of our selective basic block encryption architecture as a function of the block select rate. The increased nonsequential access time for the FPGA has the expected effect on the instruction cache miss cycles, which for several of our benchmarks become a more significant portion of the overal execution profile. It is interesting to note that the average performance penalty of the case where the entire program is initially encrypted is less than 20%, a number that is significantly less than the seemingly simpler register-sequence encoding approach.

Why is this the case? This question can be answered by again examining the ratio of the instruction cache miss cycles to the pipeline execution cycles. Although the AES instantiation brings with it a significant increase in the former factor when compared to the register-based approach, the compiler pass that encrypts the executable only requires

that two instructions (the start and stop message instructions) be inserted for every basic block. Consequently for applications that have excellent instruction cache locality such as our selected benchmarks, we would expect the performance of this scheme to be quite similar to that of the previous approach configured with a sequence length value of 2. A quick comparison of Figures 5.6 and 5.7 shows this to be the case.

These results clearly demonstrate the flexibility of our approach. With the modification of a few compiler flags an application developer can evaluate both a tamper-resistant register encoding system that covers the entire application with a significant performance detriment, to a partially covered cryptographic solution that has a much more measured impact on performance. Many of the more severe performance numbers are fully expected based on previous work - as an example, estimates for the XOM architecture [**54**] predict less than a 50% performance impact when coupled with a modern CPU. While the results in this section show that the instruction miss penalty isn't a dominant factor, this will generally not be the case when considering the larger applications that could be used with our approach. This, combined with the possiblility that programmable FPGA resources will not all be allocated for software protection purposes motivates the configuration of these resources as performance-oriented architectural optimizations. We examine instruction prefetching in the following section to investigate the impact of such an approach.

5.3. Exploring Hardware and Software Optimizations

In many cases, it would make sense for the application developer to select the basic blocks to apply the register code insertion or block encryption techniques on an individual basis. Appropriate targets for such a selection approach would be those which are most

likely to be attacked - examples being basic blocks that verify for serial numbers or that check to see if a piece of software is running from its original CD. However, it is also useful to cover a greater percentage of the entire executable with such a technique, as doing so would both increase the confidence that no block could be altered and would hinder the static analysis of the vital protected areas.

5.3.1. Intelligent Basic Block Selection

In the previous section a random select approach was investigated, where it was discovered that the performance penalty of the register code insertion technique was not often severe even in the case where all eligible basic blocks were encoded. However, it makes sense to develop more intelligent algorithms for basic block selection, as this would allow the embedded application provider more flexibility in terms of positioning on the performance/security spectrum. Before developing these techniques it is necessary to first identify the root causes of the increase in total execution time.

Assuming a constant FPGA access latency, the key source of performance degradation for our approach is the addition of instructions needed to form register sequences of the desired length. Given a basic block i of length l^i_{bblock} and a requested sequence length of l_{key} registers, the transformed basic block length can be written as:

$$l^{i'}_{bblock} = l^i_{bblock} + \alpha^i \cdot l_{key}\,, \tag{5.7}$$

where $\alpha^i \in [0, 1]$ is the percentage of a requested sequence length that cannot be encapsulated using register allocation. For example, for a register sequence length of 8, a basic block that can hide 6 of the needed key values in its original instructions would have

an α^i value of 0.25, meaning that 2 additional instructions would need to be inserted to complete the sequence.

The number of extra pipeline cycles required by the execution of the inserted instructions is highly dependent on the loop structure of the original basic blocks. As an example, an encoded basic block that is executed $O(n^2)$ times will have a considerably greater performance penalty when compared to a neighboring block that is executed only $O(n)$ times. Consequently special consideration should be given to high-incidence blocks (i.e. blocks that are deeply nested in loops). Assuming that each basic block i is executed n^i_{iters} due to its containing loop structure, the total delay due to the execution of i can be estimated as:

$$t^i_{delay} = n^i_{iters} \cdot l^i_{bblock} \cdot CPI^i \,, \tag{5.8}$$

where CPI^i is the average number of cycles per instruction required of the instructions in basic block i. This delay value ignores the specific types of instructions inside the basic block since this variation is actually quite small. If basic block i is selected for encoding using the register sequence technique, the new total execution time can be estimated as:

$$t^{i'}_{delay} = n^i_{iters} \cdot \left(l^i_{bblock} + \alpha^i \cdot l_{key}\right) \cdot CPI^{i'} \,. \tag{5.9}$$

This equation shows that an obvious approach in selecting basic blocks for encoding is to sort by increasing number of iterations. In practice, however, there will be several blocks in an application that will not be fetched at all during normal program execution. Applying dead code elimination will not necessarily remove these blocks, as they are quite often used for error handling or other functionality that is rarely needed but still vital.

For this reason the blocks where $n^i_{iters} = 0$ are placed at the end of the ordered list of eligible basic blocks.

This can only be a partial solution, as it will be very likely that ties will exist between different basic blocks with the same n^i_{iters} value. How should these blocks be ordered? The simplest method would be to order these subsets of blocks randomly, but a better approach can be found by further examining the impact on performance of the added register sequence instructions. A suitable heuristic can be developed by considering that a basic block with a larger l^i_{bblock} value will have a relatively smaller number of additional instruction cache misses after a register sequence encoding when compared with other blocks with the same n^i_{iters} value but with a shorter original block length. For this reason blocks can be effectively sorted first by non-zero increasing n^i_{iters} values and then by decreasing l^i_{bblock} values.

To estimate the number of iterations for the basic blocks in the selected benchmarks, profiling runs were performed that fed back the individual loop counts to the basic-block selection module of the SAFE-OPS compiler. As this approach can potentially lead to a significant increase in compile time, the profiling experiments were ran using smaller input sets for the benchmarks, with the goal of not letting profiling information increase the compiler run-time more than 6 times. This increase is acceptable. Embedded systems can tolerate much larger compilation times than their general-purpose counterparts since the resulting programs are used so many times without further compilation.

Figure 5.8 shows the general trends seen when the tests of the previous section are re-run with the iteration-based block selection policy. These results show that by initially selecting blocks with low iteration counts, the large performance degradations can be

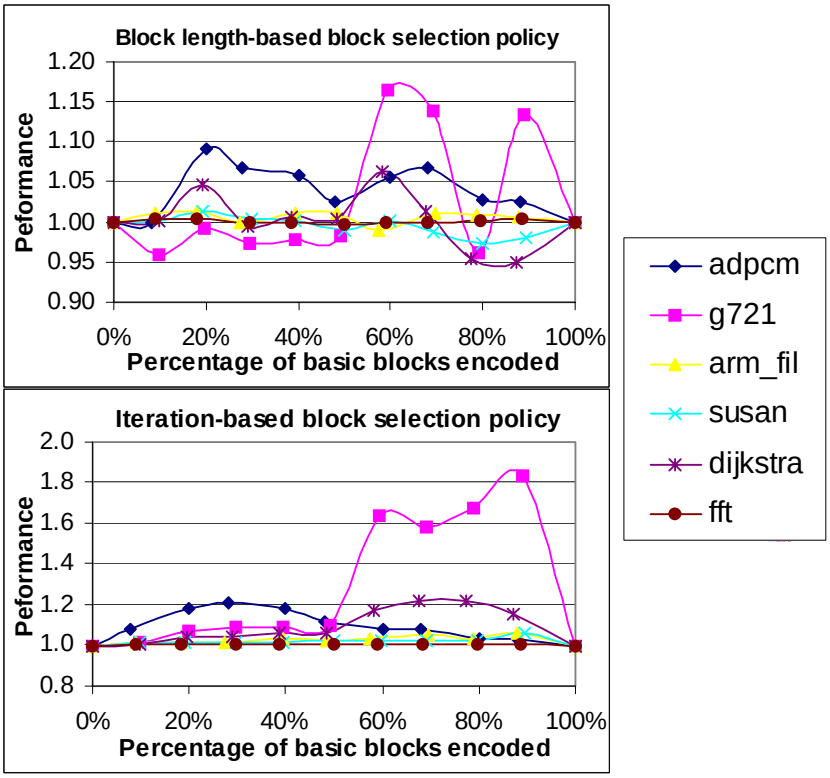

Figure 5.8. Performance characteristics of the iteration-based block selection policy. The top graph shows the absolute performance impact that can be visually compared to Figure 5.4. The bottom graph shows the performance improvement relative to the random selection approach. These results show that by avoiding basic blocks that iterate excessively, the iteration-based policy clearly outperforms the random approach, in some cases by as much as 80%.

delayed until only the highest level of security is required. As can be seen from the figure, the performance gains due to intelligent basic block selection are as large as 80% in one case (*g721*), average between 5-20% for the majority of the benchmarks, and is negligible for the *fft* benchmark. The reason the *g721* benchmark performs so well is that the code size is relatively small, and that an overwhelming percentage of the total run-time is concentrated in a just a few basic blocks. Conversely, the *fft* code size is relatively large,

and much of the run-time is spent in mathematical library calls, instructions that are not eligible for encryption under the current approach. As is the case when no blocks are selected, when 100% of the blocks are targeted for encryption using any of the techniques the performance is the same; in these cases it is more interesting to look to architectural optimizations for a source of performance speedups.

5.3.2. FPGA-based Instruction Caching

Given the reconfigurable nature of FPGAs, it is interesting to explore architectural optimizations that could be implemented to improve performance, while maintaining a desirable level of security. This section investigate using the FPGA as a program-specific secondary level of instruction cache. Three specific techniques are examined:

- *L2 cache.* As a comparison to the more application-specific caching techniques listed below, a 32 KB L2 cache was instantiated on the FPGA. As this would consume a considerable amount of programmable resources on the FPGA, this could be implemented using a relatively fast on-chip FPGA memory and therefore in this configuration the access latency requirement was increased to 8 cycles.
- *Basic block cache.* In this configuration, the FPGA caches only the instructions that are fetched inside selected basic blocks. After the entire block is validated, this smaller cache can then return the instruction values for future requests without requiring an expensive access to main memory or any other cycles spent in decryption. This technique would gain in preference under a configuration where the FPGA decryption algorithm requires a relatively large number of computational cycles.

- *Register sequence buffer.* The FPGA in this architecture is able to determine which register sequence values are original instructions, and which are instructions that are inserted by the compiler to complete the sequence. This is possible with the addition of a fairly small amount of instruction decode logic. While a selected block is being fetched and validated, the FPGA stores the relative locations of the inserted register codes in a simple buffer. In future requests for these validated basic blocks, the FPGA can return NOP instructions instead of fetching instructions at these register code locations. This architecture is the simplest of the three in terms of required resources, requiring a buffer of at most $l_{key} \cdot n_{bblock}$ bits.

The common feature among these three approaches is that after the FPGA fetches an instruction from main memory and validates it, it then attempts to store the validated instruction in a faster buffer.

In evaluating these architectures the effect on performance for three benchmarks was examined when the sequence length was kept constant at 8 and the percentage of basic blocks to be selected (randomly) was increased from 0-100%. Figure 5.9 shows a summary of the instruction memory hierarchy performance results of these techniques, normalized to the case where no architectural optimizations are applied. These results clearly show that the L2 cache performs much better than the other two approaches. As can be seen from the figure, instantiating an L2 cache on the FPGA leads to performance speedups of as much as 55% in some cases, as compared to the basic block caching approach which is limited to a 5% speedup and the register sequence buffer which demonstrates less than 3% performance improvement. This is to be expected, as the relatively large

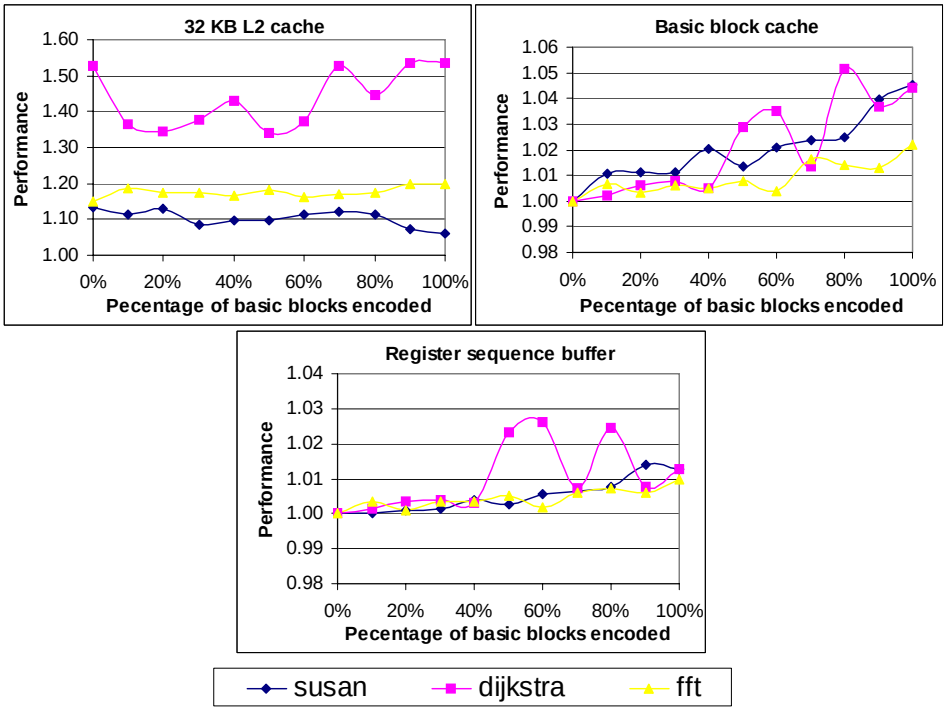

Figure 5.9. Performance of the three architectural optimizations, relative to the base case when no instructions are cached in the FPGA. These results show that while the L2 cache instantiation performs by far the best of the three, the architecturally much simpler basic block caching technique slightly improves performance when the percentage of encrypted basic blocks approaches 100%.

cache allows us to buffer a high percentage of all instructions in the benchmarks. While the L2 cache technique performs the best overall, it is interesting to note that the basic block cache and register sequence buffer approaches show near-linear speedups as the aggressiveness of the encryption is increased. This is an important result, as it implies that if the other system parameters are tuned for more cryptographic strength (i.e. by increasing the key length, implementing more complex instruction filters, implementing more algorithmically advanced register sequence translators, etc.), these two software

protection-specific approaches will begin to perform favorably well when compared to the general L2 cache case.

When evaluating the implementation of architectural-based optimizations on the FPGA it is necessary to consider that the clock speed of the FPGA is dependent on the critical path of the instantiated logic. In order to effectively measure the performance of an FPGA architectural technique, it is important to consider configurations of varying clock speeds. Figure 5.10 shows the results of such experiments on the *susan* benchmark. In gathering these results, the register sequence length was held constant at 8 and the percentage of encoded basic blocks constant at 25%. From this figure it can be clearly seen that as the clock rate of the FPGA is reduced via simulation, the performance benefits of the L2 cache implementation can quickly turn negative. This drastic effect on instruction memory hierarchy performance is a function of the additional FPGA cycles required by the architecture. This is due to the fact that the memory bus speed is kept constant, and any cycles spent by the FPGA grow in importance as its clock rate decreases. These results show that if the L2 cache implementation required a clock rate of 150MHz or slower, the basic block cache and register sequence buffer techniques would perform better, assuming their minimal logic would allow them to continue operating at 200MHz.

5.4. FPGA-based Instruction Prefetching

For systems with a low instruction cache efficiency and long memory latencies, both buffering and prefetching have been shown to be useful optimizations for reducing cache misses and subsequently improving system performance. In this section we concentrate on

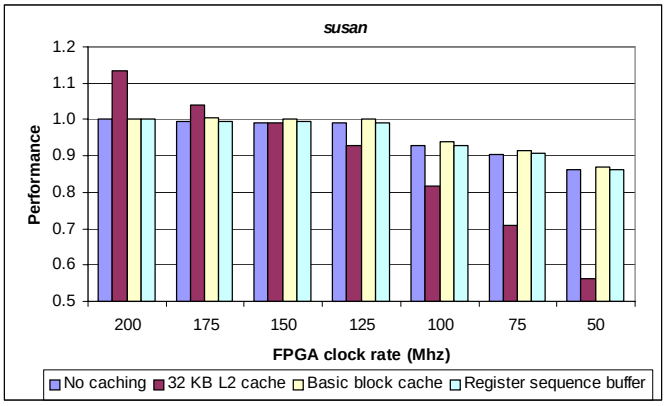

Figure 5.10. Performance of the three architectural caching strategies for varying FPGA clock rates. These results demonstrate that the resource-utilization of the FPGA needs to be taken into account when considering architectural optimizations, as the strategy that performs the best under the assumption of an ideal clock rate (L2 cache), ultimately performs the worst given the condition that the clock rate would need to be lowered to 50MHz to compensate for the additional applied logic.

prefetching and examine how including instruction prefetching logic on the FPGA affects performance.

Prefetching techniques tend to monitor cache access patterns in order to predict which instructions will be requested in the immediate future. These guesses are used as the basis whereby instructions are fetched from slow main memory into cache before they are requested. In general, the usefulness of a prefetching scheme is based on the percentage of cache misses that it is able to prevent. In other words, prefetching instructions will only improve performance if the average per-access overhead of the prefetching hardware is outweighed by the average overall fetch latency.

One of the most popular prefetching techniques is known as *next-N-line* prefetching. The main concept behind this technique is that when a block of instructions are requested main memory, the prefetching mechanism also returns the next N lines to the instruction

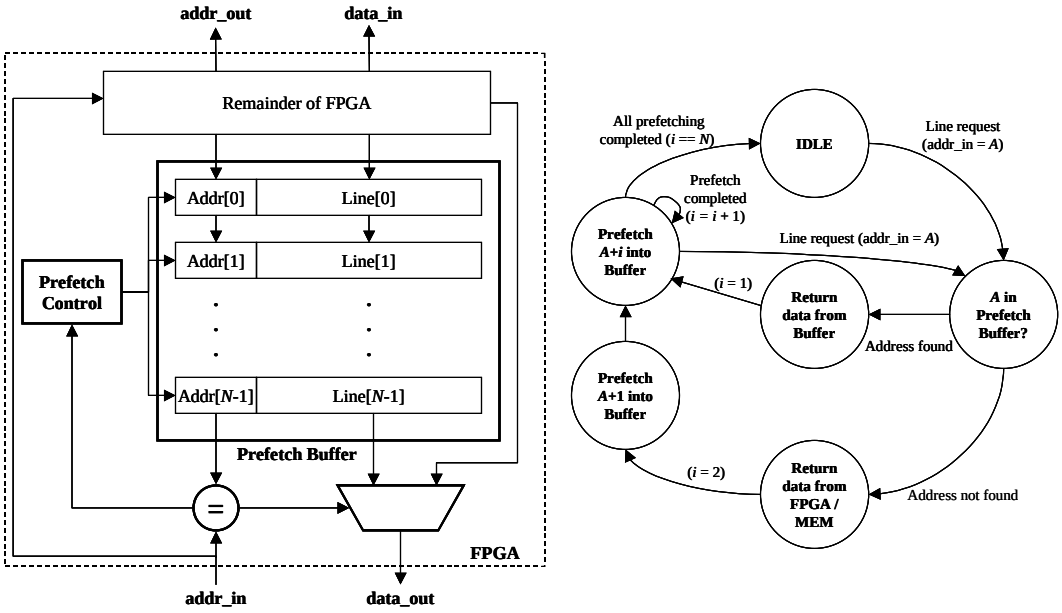

Figure 5.11. Architecture and state diagram for a next-N-line prefetching technique implemented in FPGA

cache. Our architecture for next-N-line prefetching can be described as follows (see Figure 5.11). A relatively small N-line buffer is instantiated on the FPGA which responds to instruction cache requests concurrently with the other security-related mechanisms of the FPGA. If the requested line is found, then the other FPGA operations are canceled and the data is returned directly from the buffer. If there is no match, then the result is returned from higher-level memory with no additional delay (assuming the latency in accessing higher-level memory is greater than that of the prefetching buffer). In either case, the next N lines following the requested address are then fetched through the remainder of the FPGA. This can be a quite simple operation for sequentially fetched blocks - it is very likely that many of the next N lines have already been recently prefetched and do not need to be fetched again. In order to minimize the average prefetching overhead

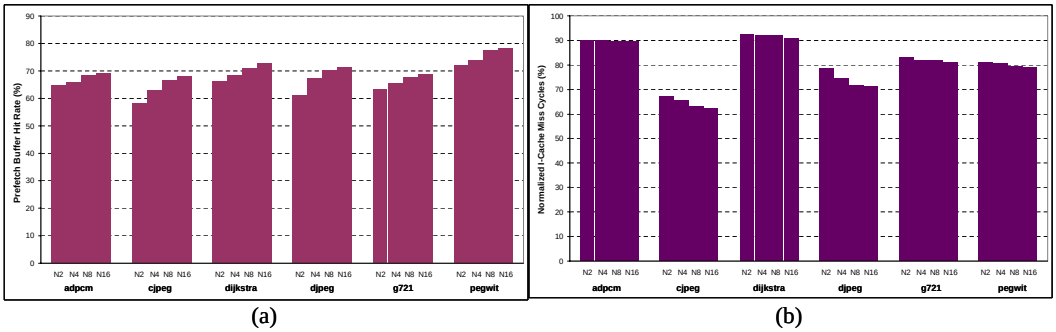

Figure 5.12. Effectiveness of next-N-line instruction prefetching optimization as a function of N. These results show that (a) a 16-line buffer can successfully cache on average about 71% of the requests, although the majority of those hits can be had in a much smaller buffer. (b) These hits lead to on average a 18% savings in instruction cache stall cycles.

we configured our FPGA to halt its current prefetching operation in the case of a new request from the L1 cache.

It is important to note that this requires no changes to the L1 instruction cache itself. This is nice feature as it means that if a cache miss leads to a poorly speculated prefetch operation (as could be the case in a procedure call that quickly returns to the callee location), the cache will not get polluted with useless lines. Also since our prefetch control is configured to not continue in its fetching of the next N lines in the case of a new request, such an architecture is likely to service any given request with a minimal additional delay.

We configured the FPGA in our selective basic block encryption approach to perform next-N-line instruction prefetching for the case when 25 percent of the eligible basic blocks are selected. The results for varying values of N can be seen in Figure 5.12. Our most aggressive prefetching architecture (N = 16) does a fairly good job at predicting future misses - on average the prefetch buffer in this configuration successfully responds to 71% of

the instruction cache requests. This correlates directly to an instruction cache miss cycle improvement. Figure 5.12b shows that on average there is an average of 18% savings in instruction stall overhead. For our current benchmarks this translates to a relatively small overall performance improvement but given some benchmarks with a lower instruction locality this can have a great effect with no additional hardware cost.

CHAPTER 6

Ensuring Program Flow Integrity

Embedded applications are typically not the main focus of secure solutions providers, as the personal and business computing world has been the traditional target for wide-scale attacks. This is a cause for concern for two reasons. Firstly, embedded software tends to be written in low-level languages, especially when meeting real-time constraints is a concern. These languages tend to not have facilities for runtime maintenance which can often cover programming errors leading to security holes. Secondly, embedded systems are used in a variety of high-risk situations, for which a malicious attack could have devastating consequences.

Many current approaches merely apply stopgap measures - either by patching specific vulnerabilities or disallowing behavior that is representative of the attacks considered. While effective at preventing the most popular exploits, these approaches are not able to handle an unexpected weakness or an unknown class of attacks. Clearly, additional research is needed to find a more general solution.

In this chaper, we present a compiler module and reconfigurable that can prevent a more general class of attacks. The input program is first analyzed by our compiler to generate program flow metadata. When the application is loaded onto the processor, this data is then stored in the custom memories on our architecture. Since this approach requires only a passive monitoring of the instruction fetch path, we are able to continually verify the program's execution with a minimal performance penalty.

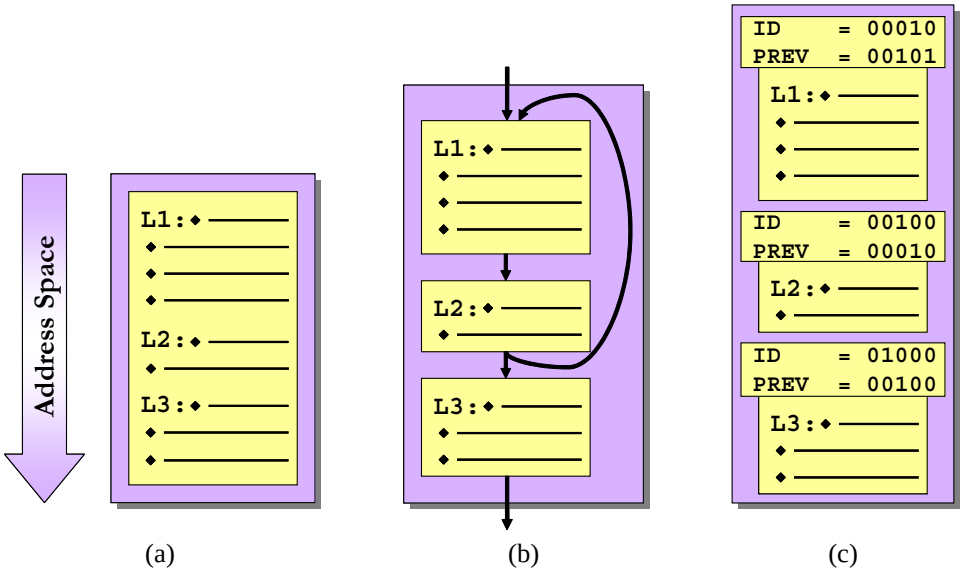

Figure 6.1. (a) Static view of an application's instructions (b) Program-flow graph (c) Runtime execution information added to the application via the PFencode compiler module

6.1. The PFcheck Architecture

When considering a static view of an application (Figure 6.1), a *basic block* is defined as a subsequence of instructions for which there is only one entry point and one exit point. Basic block boundaries are typically found at jumps and branch targets, function entry and exit points, and conditionally executed instructions. The various types of program flow attacks can all be generalized as invalid edges in a flow graph at the basic block granularity:

- Flow passing between non-consecutive instructions in the same basic block.
- Flow passing between between two blocks with no originally intended relation.
- Flow passing between any two instructions that are not both at basic block boundaries.

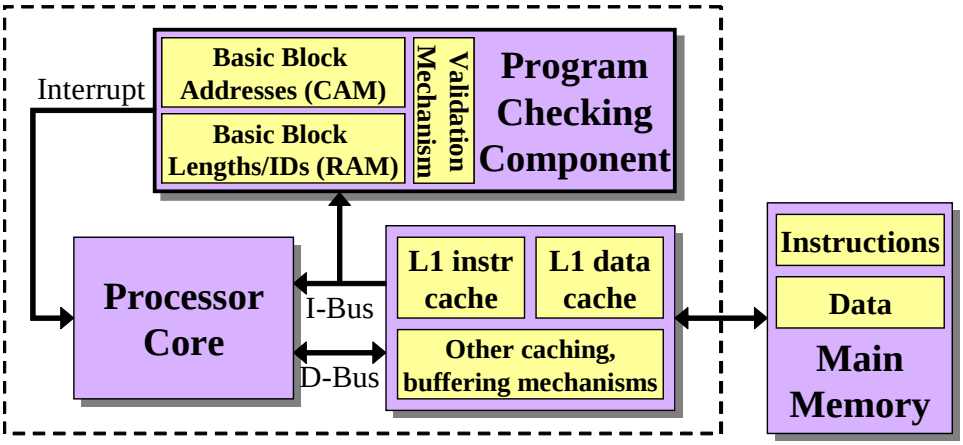

Figure 6.2. Insertion of the PFcheck component into a standard CPU architecture

- Flow passing from an instruction back to that same instruction, when that instruction is not both the entry and exit point of a single basic block.

Our approach operates at the basic block level. In the example of Figure 6.1, flow passes unconditionally from block `L1` into `L2`, where it may either loop back into `L1` or pass conditionally into `L3`. This information is statically knowable by the compiler, which is typically able to break the input program into basic blocks for analysis. It is also possible to include a profiling pass to determine more complex program flows. Our compiler module, which we call PFencode, assigns identification labels to the basic blocks. These "IDs" are given a one-hot encoding, for reasons that will become apparent. For this example, block `L1` is encoded as **00010**, `L2` is encoded as **00100**, and `L3` is encoded as **01000**.

Next, the compiler generates a table of predecessors for each basic block. Since the blocks are already encoded with a unique bit, we can create this "prevID" table by just ORing the IDs or all the blocks that are valid predecessors for each block. As an example, since since block `L2` and the prefix code segment (with an ID of **00001**) are both valid

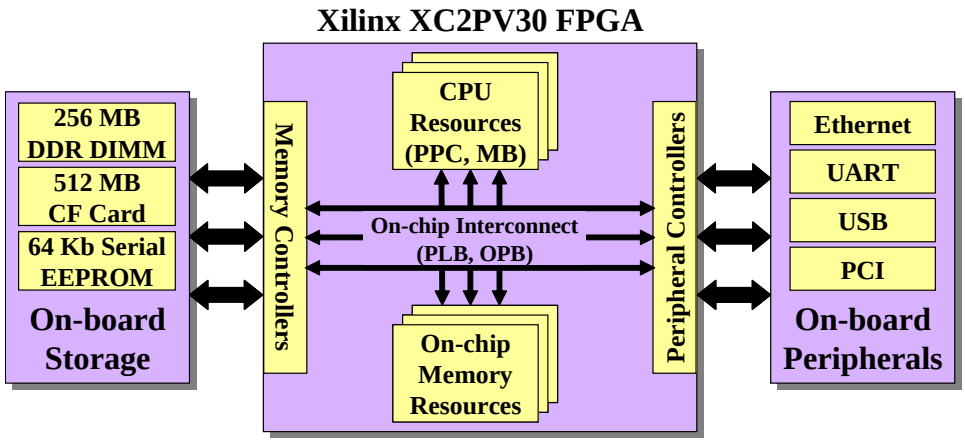

Figure 6.3. Architectural view of the Xilinx ML310 development platform

predecessors of `L1`, the prevID value for `L1` is defined as **00101**. The compiler also generates a value representing the length of each basic block.

Figure 6.2 shows a high level view of how our program flow checking architecture can be inserted into a standard CPU architecture. Our hardware component, which we call PFcheck, uses a snooping mechanism to analyze instructions being fetched directly between the processor core and the lowest level of instruction cache. Since the PFcheck architecture does not delay the instruction fetch path, it is not expected to incur the same negative performance impact as other approaches.

The architecture contains several customized memories which it uses to hold the program flow metadata values. The basic block base addresses are stored in a Content-Addressable Memory (CAM). Typically on a CAM lookup operation, the output is a unencoded value of the lines in the memory that match the target address. When an instruction is being fetched, the PFcheck component sends this address to the CAM. If there is a match, this means that the instruction is an entry point for a basic block - the ID of that block is given as the output of the CAM.

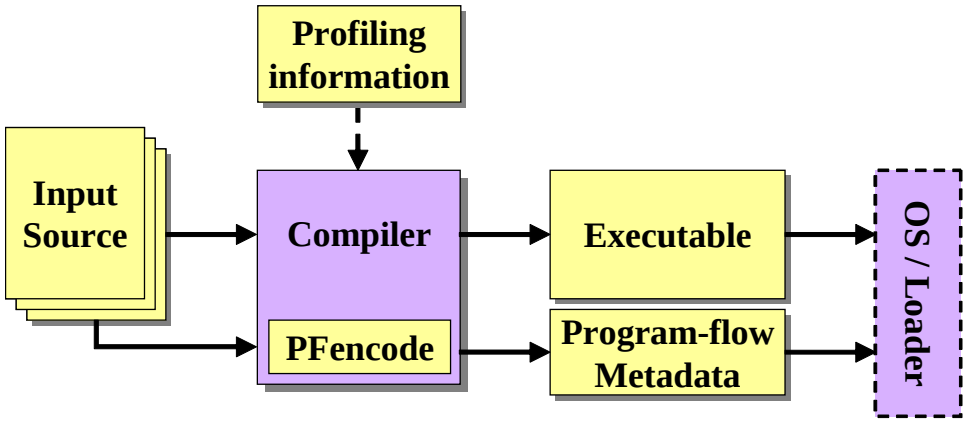

Figure 6.4. Addition of the PFencode module into the GCC compiler

The basic block lengths and prevID table are stored in a standard RAM. The only output required by the component is an interrupt signal, which is used to notify the system that an invalid flow has been detected. A more detailed description of a PFcheck implementation is given in the following section.

6.2. Implementation and Results

We implemented our prototype system on a Xilinx ML310 FPGA development board (Figure 6.3), using version 7.1 of their Embedded Development Kit (EDK) for the design entry, simulation, and synthesis. The ML310 contains a XC2VP30 FPGA that has two PPC 405 processors, 13696 reconfigurable CLB slices, and 136 Block SelectRAM modules. The board and FPGA can act as a fully-fledged PC, with 256 MB on-board memory, some solid-state storage, and several standard peripherals.

We implemented our PFencode compiler (Figure 6.4) as a post-pass of the GCC compiler targeting the Xilinx MicroBlaze soft processor [**95**]. We chose the MicroBlaze over

Table 6.1. PFencode results for a variety of benchmarks

Benchmark	App Size	Num Basic Block	Metadata Size	Percentage Increase
`adpcm`	15.8 kB	75	1.30 kB	8.2%
`dijkstra`	23.3 kB	257	10.3 kB	44.2%
`laplace`	15.6 kB	32	0.38 kB	2.4%
`fir`	22.8 kB	240	9.12 kB	40.0%
`susan`	90.8 kB	1253	206 kB	226.9%

the PPC for its simplicity and flexibility. Table 6.1 shows the results when we ran PFencode on a number of benchmarks customized to support MicroBlaze software libraries. For these relatively small applications, we found that the size of the metadata increased exponentially with the number of basic blocks. This is due to the fact that for an application with N basic blocks, our one-hot encoding approach requires an $N \times N$ memory to hold the entire prevID table. For the benchmarks that have relatively short basic blocks, this increase in application size can be drastic (see `susan` which is 5.8 times the size of `laplace` but requires 542 times the amount of metadata).

Figure 6.5 shows the internals of our implementation of the PFcheck architecture. The instruction addresses (M bits wide) is sent into the CAM module which has M-bit wide entries for each of the N basic blocks. If there is a match, that means that this instruction address is a basic block base address - the one-hot ID value for that basic block is the output of the CAM on a match. Otherwise, the output of the CAM is all zeros indicating that the current instruction is inside a basic block. The output of the CAM is sent to an encoder that creates a $log(N)$ wide signal that drives the address of the RAM module to get the prevID value for the current block. The entry in the prevID table is checked with the previously fetched ID that is stored in a register. If the bits do not match up, an interrupt signal is sent to the processor. In the case where the current instruction is not

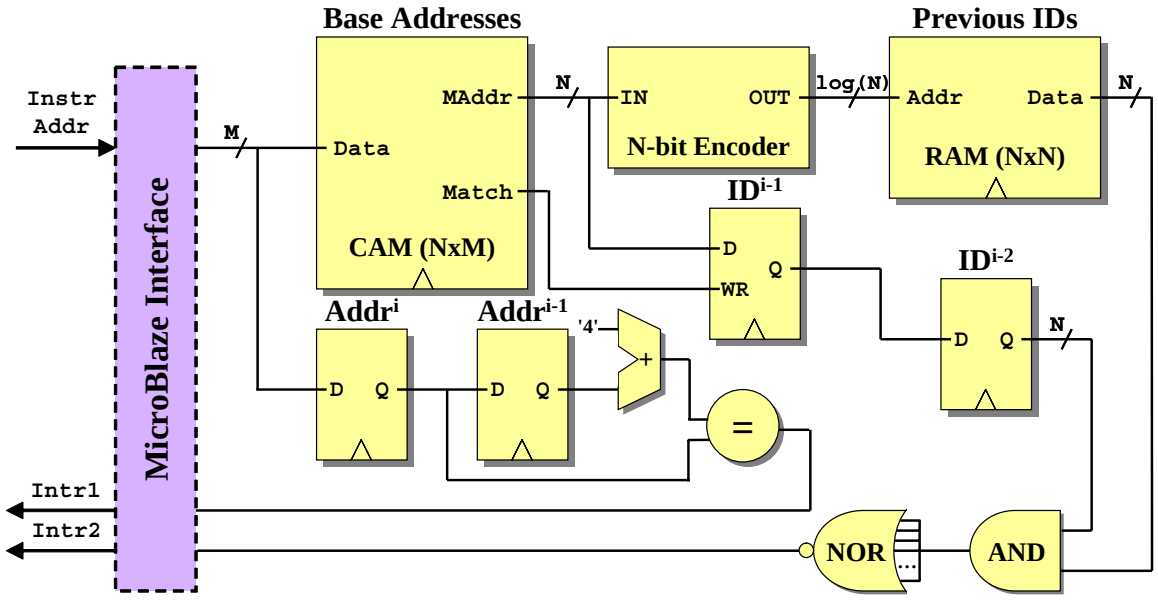

Figure 6.5. Internals of an implementation of the PFcheck architecture

a basic block entry point, an interrupt will be also be generated if the current instruction does not directly proceed the previous instruction. This is equivalent to checking that PC does not equal $PC + 4$.

Table 6.2 shows how the area consumption and performance of the PFcheck architecture is depends on the configuration. For our experiments, we synthesized designs with varying amounts of basic block support and address bus width. Each design is labeled $N \times M$. Several trends are readily apparent. The BlockRAM and slice usage increase linearly with N. Decreasing the address bus width has less of an impact on area usage, and no impact on clock frequency. The limiting factor for this medium-sized FPGA was the number of BlockRAMs; we were not able to fit the `1024x32` on the device.

To test for correctness, we used PFencode to compile an application that contained a buffer overflow vulnerability. We attempted to write attack code on the stack and to

Table 6.2. Area and performance results for a PFcheck implementation on a Xilinx XC2VP30 FPGA

Config ($N \times M$)	64x16	64x32	128x32	256x32	512x32
Num Slices	450 (3.2%)	510 (3.7%)	840 (6.1%)	1615 (11.8%)	3031 (22.1%)
Num BRAMs	5 (3.7%)	9 (6.6%)	18 (13.2%)	36 (26.5%)	80 (58.9%)
Clock Freq	71.3 MHz	71.3 MHz	64.9 MHz	56.0 MHz	59.32 MHz

overwrite the return address to point back to this buffer. While the architecture did not protect values on the stack from being overwritten, when the program attempted to return to this malicious region an interrupt was triggered.

Further improvements are being made to the PFcheck architecture. We are currently analyzing the performance overhead of writing the program flow metadata values to the CAM and RAM. Also, we are investigating a paging architecture that will be able to protect larger applications (in terms of number of basic blocks) without a loss in the amount of security provided.

CHAPTER 7

Optimizing Symmetric Block Cipher Architectures

Cryptography is a core component in most secure systems [**58**] and is used extensively to provide secrecy, integrity, and authenticity. The widespread deployment of cryptographic methods and their typically intensive computational requirements motivate the development of techniques to maximize their performance.

Field-Programmable Gate Arrays (FPGAs) have become an attractive target for cryptographic routines, as witnessed by the abundance of commercial and academic implementations of private-key ciphers [**28, 39, 47, 49, 71**], public-key ciphers [**29, 59, 66**], and secure hash algorithms [**35, 55**]. This increased interest in reconfigurable logic can be attributed to the following factors:

- Since the individual operations of a typical cryptographic algorithm are simple (often consisting only of combinational logic), specialized hardware designs can greatly reduce the overhead associated with an equivalent software implementation.
- Dedicated hardware can further improve performance relative to software designs by extracting functional parallelism from cryptographic algorithms.
- The nonrecurring engineering costs associated with FPGA development typically are significantly lower than those associated with full-custom ASICs, making FPGA development very attractive for all but the highest volumes of production.

- FPGAs are also an attractive alternative to ASICs in that the FPGA design flow does not require the designer to perform tasks such as clock tree generation, boundary scan insertion, and physical verification, all of which are essential steps in the ASIC design process.
- The reconfigurable nature of FPGAs allows the modification of an implementation at any point during, or even after, the design process [**91**]. Applications of this feature include fixing flaws discovered in the algorithm after design time, optimizing the architecture for a known range of inputs [**34**], and switching between a set of ciphers at runtime [**27**].

Much of the recent work in this area has focused on high-throughput or low-area implementations of Symmetric Block Ciphers (SBCs). An SBC encrypts data by transforming plaintext blocks of a fixed length into ciphertext under a secret key, and uses the same key to transform ciphertext blocks back into plaintext during decryption. As illustrated by Fig. 7.1, a typical SBC algorithm consists of an iterative series of data transformations called a round function. A typical round function consists of simple transformations such as bitwise logical operations, arithmetic operations, and permutations; often, round functions also include a set of non-linear byte substitutions, which are most commonly represented as a lookup table known as a substitution box (S-box). Examples of SBCs include the Data Encryption Standard (DES) [**31**] and the Advanced Encryption Standard (AES) [**30**].

In order to encrypt messages longer than a single block, various modes of operation can be applied to SBCs [**1**]. Electronic codebook (ECB), the simplest mode, breaks the input message into blocks and processes each block independently under the input key. Under

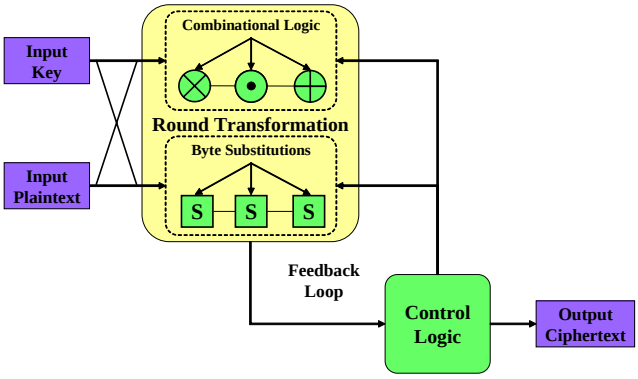

Figure 7.1. A generic Symmetric Block Cipher (SBC) encryption architecture

ECB, identical plaintext blocks within a message are encrypted to identical ciphertext blocks, which is often an undesirable behavior. As a result, modes incorporating some sort of feedback or key agility are often preferred for the encryption of longer data streams.

Compared to other efforts in implementing SBC architectures on FPGAs, our work is unique in that we combine previous attempts at extracting fine-grained parallelism from individual cipher operations with techniques that leverage a coarser parallelism and make use of the large quantities of resources available in modern FPGA devices. Another key distinction stems from our top-down design methodology (Fig. 7.2) that allows for area and delay tradeoffs to be managed at several levels of the design hierarchy using a single parameterizable core. In doing so, we draw closer to the goal of maximizing the potential of FPGAs for accelerating the performance of SBCs. In this chapter, we focus on AES encryption and explore the area, delay, and efficiency tradeoffs associated with a high-throughput FPGA implementation. We present a design that is capable of encrypting 128-bit data blocks at speeds greater than 30Gbps.

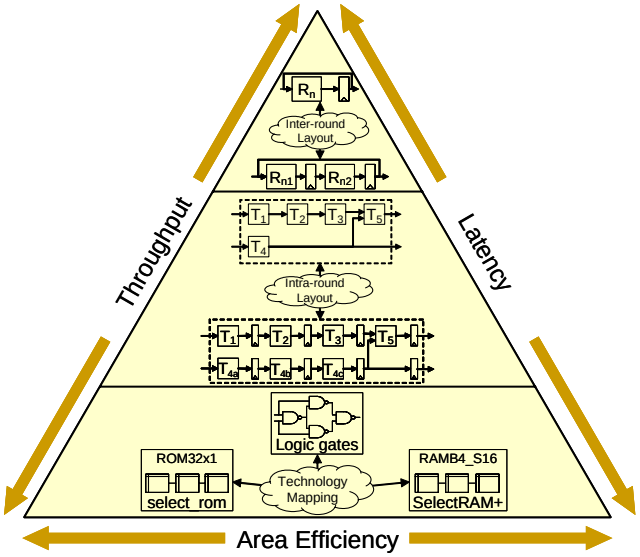

Figure 7.2. Top-down block cipher design methodology

7.1. AES Overview

In 1997, the U.S. National Institute of Standards and Technology (NIST) held an international competition to replace DES as the federal information processing standard for symmetric cryptography. The fifteen designs submitted to NIST were publicly evaluated based on several factors including simplicity, security, and suitability to both hardware and software implementations. In 2000, NIST selected Vincent Rijmen and Joan Daemen‘s Rijndael algorithm as the winner of the AES competition.

As an SBC, AES operates on blocks of a fixed length, uses the same key for encryption and decryption, and is amenable to various modes of operation. Though AES and Rijndael are often used interchangeably in practice, the two are not actually identical. Whereas Rijndael supports various block and key lengths, AES is restricted to 128-bit blocks and key lengths of 128, 192, and 256 bits. Based on the observation that the algorithms for

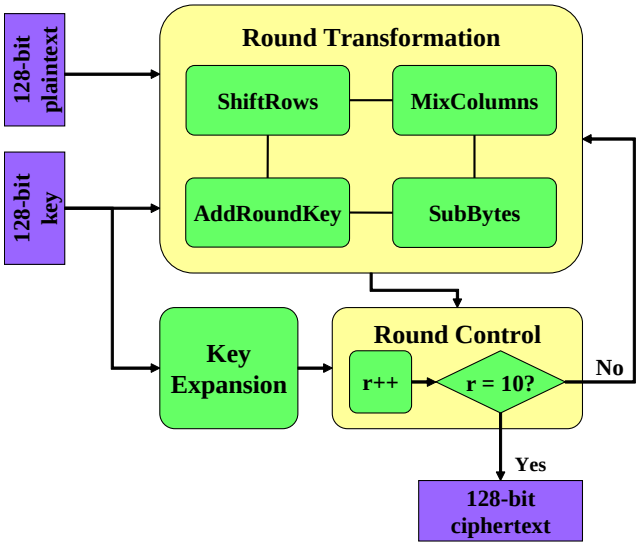

Figure 7.3. Algorithmic view of AES-128E

encryption and decryption using various key lengths are very similar, the remainder of this paper focuses on AES encryption using a 128-bit key (AES-128E) with only a minor loss of context compared to examining every key length supported by AES.

The structure of AES-128E is shown in Fig. 7.3. The encryption operation takes, as inputs, a secret key and a 128-bit plaintext block. The secret key is expanded into individual sub-keys for each round by the `KeyExpansion` function, while the input plaintext, represented as a 4x4 byte array, forms the initial state of the operation. Based on the current state and sub-key, each round generates a new state, which is calculated through the following four steps:

- `SubBytes` replaces each byte in the state with another, according to a non-linear substitution. As previously mentioned, this step is performed using an S-box.
- `ShiftRows` rotates each row of the state by a certain number of byte positions.

- `MixColumns` performs an invertible linear transform on each column of the state, such that each byte of the input affects all four bytes of the output.
- `AddRoundKey` combines the state with the sub-key of the round, using a bit-wise XOR operation.

A more detailed description of the AES algorithm is available in the official AES standardization documents [**30**] and the cipher developers own publications [**25**].

7.2. Parallelizing the AES Cipher

As illustrated by Fig. 7.4, a typical SBC applies a cryptographic function $E_K(P)$ to an input key K and input plaintext P, producing an output ciphertext C. A significant amount of parallelism can be extracted from these types of ciphers, providing the potential for greatly improved performance. In this section we describe several SBC parallelization techniques which are readily applicable to AES. We also examine the suitability of each technique for FPGA implementations in terms of performance and resource utilization.

7.2.1. Coarse-grained Parallelism

The algorithm describing a cipher operating in ECB mode is often embarrassingly parallel. A large plaintext input P can be broken into smaller pieces P^1, P^2, P^3, etc, each of which can be processed concurrently and independently using multiple identical resources. We refer to this property as coarse-grained parallelism. In the case of FPGA implementations, a large input can be broken into smaller pieces and issued to several identical computational units for simultaneous processing. The output of each of these units can then be recombined to form the output of the cipher. This coarse-grained parallelism

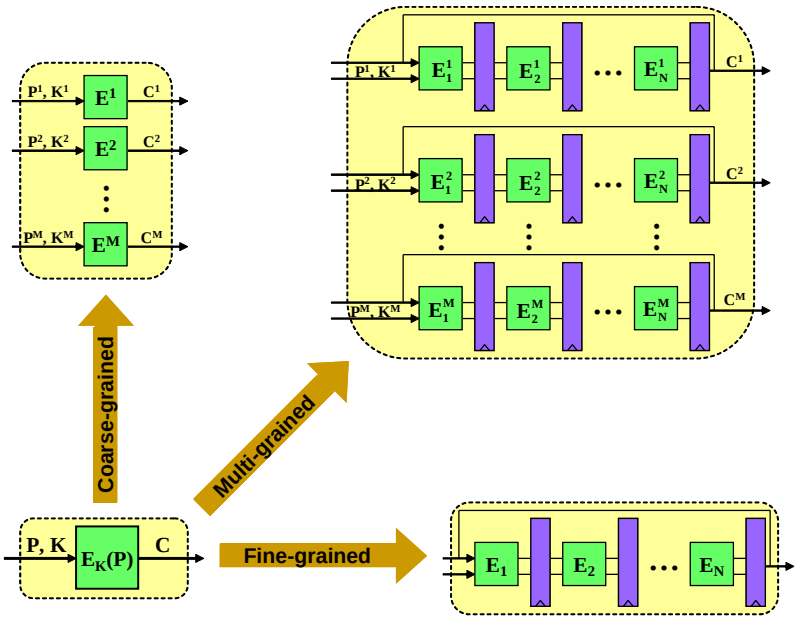

Figure 7.4. Coarse, fine, and multi-grained parallelism in SBC architectures

optimization, which we refer to as *cipher replication*, requires the original cipher to be replicated M times (Fig. 7.4).

As previously mentioned, ECB mode is often not suitable for applications requiring the encryption of large streams of data; however, this observation does not eliminate the usefulness of cipher replication. Applications intended to support the simultaneous processing of multiple independent data streams, such as network processors, can still benefit from cipher replication by assigning each replicated cipher an independent data stream.

Given an FPGA will sufficient resources, this approach would theoretically lead to linear increases in throughput without adverse effects on latency or area efficiency. As demonstrated in Section 7.4, however, this is often not the case, primarily due to the fact that the dramatic increases in reconfigurable logic usage associated with this optimization

greatly complicate interconnect routing. This idea of replicating block cipher architectures is inspired by Swankoski et al. [**78**] and is something of an inverse concept to the multi-threaded architecture proposed by Alam et al. [**4**].

7.2.2. Fine-grained Parallelism

As many SBC algorithms, including AES, are iterative structures, it is possible to extract fine-grained parallelism through the *unrolling* of round functions. Similar to the unrolling performed by a software compiler that replaces a loop with multiple copies of the loop body, unrolling the round function structure effectively replaces it with N sequential copies, where $1 \leq N \leq N_{round}$. For AES-128E, $N = 1$ corresponds to the normal iterative case, while $N = 10$ specifies a fully unrolled implementation. In practice, there is often little or no advantage to unrolling by an N that does not divide the loop count evenly. Applying round unrolling by itself potentially reduces the latency of the cipher, at the expense of clock rate and a significant increase in reconfigurable resource utilization associated with the replication of round functions.

Unrolled rounds are eligible for *pipelining*. As illustrated by Fig. 7.4, registers can be inserted between rounds, allowing new data to be processed by E_1 immediately after the previous data has been latched for use by E_2. When combined with round unrolling, pipelining allows the simultaneous processing of as many as N_{round} plaintext blocks of a single stream for a cipher operating in ECB mode, or a single block for each of N_{round} independent streams for a cipher operating in a feedback mode. The primary benefit of applying round unrolling with pipelining is a significant increase in throughput compared

to the basic, iterative form of the cipher. The downside is the added area associated with replicating round functions and inserting pipeline registers between them.

It is also important to consider the inherent parallelism within each round function. The critical path of SBCs such as AES is dependent on the series of transformations performed within the round function. Pipelining can be applied between these transformations to reduce the critical path of the cipher to that of the individual transformation with the longest delay. The clock frequency of the design can be further increased by *partitioning* slower transformations into multiple parts. We refer to this technique of extracting parallelism from individual rounds as *transformation pipelining*. The benefit of applying transformation pipeline cuts is a higher clock frequency. The costs of this technique are a small increase in area due to the added pipeline registers and often a longer latency for the cipher. The concept of fine-grained parallelism in SBC architectures is an extension of the inter-round and intra-round layout optimizations previously implemented in [**103**].

7.2.3. Multi-grained Parallelism

As indicated by Fig. 7.4, we refer to the combination of fine and coarse-grained parallelism techniques as multi-grained parallelism. The large quantity of reconfigurable resources offered by modern FPGAs provides the opportunity to combine both sets of optimizations for a very high-performance cipher. In Section 7.4 we quantitatively examine the tradeoffs associated with such an implementation.

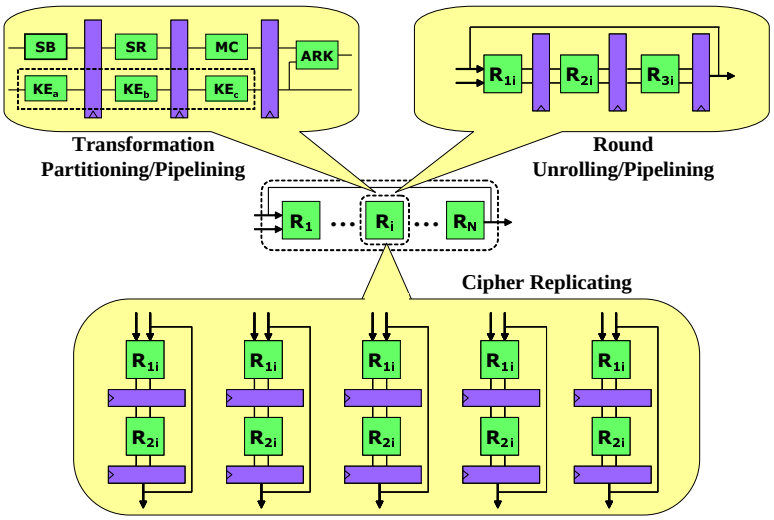

Figure 7.5. Exploiting parallelism in AES-128E: *unrolling*, *partitioning*, *pipelining*, and *replicating*

7.3. AES FPGA Implementation

For the AES implementations of this section we targeted the Xilinx Virtex series of FPGAs. Like all current Xilinx FPGAs, a Virtex device consists of an array of Configurable Logic Blocks (CLBs) and dedicated Block SelectRAM (BRAM) modules, surrounded by a perimeter of I/O Blocks (IOBs) and routed together using a programmable interconnect.

The KeyExpansion routine generates sub-keys for each round of the cipher. Since a particular sub-key is not used until the final operation of its corresponding round, it is possible to divide the KeyExpansion function into smaller, round-specific modules, each of which performs only the expansion operations required to generate the sub-key for its corresponding round. This technique, shown in Fig. 7.5, is known as online key generation.

When implemented in a pipelined fashion, online key generation is particularly useful for implementations which need to support other modes of operation, as it is possible to begin processing a new key before the current key is fully expanded. It should be

noted that if the input key is fixed over a relatively long period of time, then several pre-ciphering optimizations are possible [**34**].

The S-box lookup tables, which constitute the SubBytes operation and parts of the KeyExpansion operation, can be implemented in several ways using Xilinx technology:

- *Block SelectRAM* - the lookup table values for each S-box can be loaded into BRAM during device configuration. Since the BRAMs are dual-ported, each block can implement two separate S-boxes. BRAMs are a dedicated resource on Xilinx FPGAs, meaning that there is a hard upper limit on the number available for use in any design.
- *Distributed SelectROM* - distributed ROM primitives, pre-loaded with the lookup table values for each S-box, can be synthesized directly using CLBs. This approach typically offers better performance relative to BRAMs, but will require additional glue logic.
- *Logic* - the lookup tables can be converted to a logic representation and synthesized in CLBs. An advantage of this option is that it provides the synthesis tools an opportunity to optimize for area and delay.

Due to the number of S-boxes required for an AES-128E implementation, the choice of lookup table technology has a significant effect on the area and performance of the cipher.

The number of times that the AES-128E cipher can be replicated is limited only by the availability of resources on the target FPGA. The increase in CLB and BRAM utilization is nearly linear, as cipher replication effectively packs multiple copies of the cipher onto a single FPGA. Likewise, IOB utilization will increase linearly, as each replicated cipher needs an independent set of inputs and outputs. This heavy I/O burden quickly

becomes the limiting factor for all but the most aggressively unrolled designs. It should be noted that applications with tight I/O or board-level routing constraints can utilize data serialization methods [**93, 97**] to reduce the pin burdens associated with cipher replication. These techniques, which are already well documented, are not included in our implementation.

Since AES-128E requires 10 rounds to process an input plaintext, a fully unrolled cipher contains 10 sequential copies of the round function. While the example in Fig. 7.5 illustrates an unrolling factor of 4, we restrict our choices of partial unrolling factors to 2 and 5, in order to evenly divide the number of rounds. Depending on the chosen method of S-box synthesis, applying round unrolling techniques will dramatically increase either CLB or BRAM utilization. Once unrolled, pipeling rounds will result in a modest increase in CLB usage. Although there are many possible ways to pipeline an array of unrolled rounds, for the sake of design simplicity we do not implement partially pipelined rounds.

The simplest application of transformation pipelining inserts a pipeline register between the `ShiftRows` and `MixColumns` transformations in the round function. Since the critical path of the non-pipelined cipher lies within the round function, this approach has a direct effect on the maximum clock rate of the implementation. A more aggressive approach to transformation pipelining inserts a second pipeline register between the `SubBytes` and `ShiftRows` transformations, splitting each round into 3 stages. The reduction in propagation time associated with this approach shifts the critical path of the circuit to the round-specific `KeyExpansion` module. Fortunately, the `KeyExpansion` operation is also eligible for pipelining, as its results are not used until the last operation in the round. Consequently, combining the 3-stage round pipeline with a pipelined

`KeyExpansion` module produces further increases in maximum clock rate. Our initial experiments revealed that any additional partitioning or pipelining within the round function and `KeyExpansion` module produced no measurable benefits.

7.4. Experimental Results

The stand-alone architecture described in the previous section was implemented as a VHDL core. A configuration file was used to control the round layout, cipher replication, and S-box mapping properties of each design. The designs were synthesized using Synplify Pro 7.2.1, configured to target the Xilinx XC2V8000 device. The XC2V8000, which is the largest member of the Virtex-II family, contains 46,592 slices, 168 BRAM modules, and 1102 IOBs. Place-and-route and timing analysis were performed using Xilinx ISE 5.2i.

For each design, this toolchain was used to measure the maximum possible clock rate (f_{clk}), the number of slices (N_{slice}), and the number of Block SelectRAMs (N_{bram}). The theoretical maximum throughput of each design was then calculated by the following equation:

$$Tput = \frac{128 \cdot f_{clk} \cdot rep\_factor}{blocks\_per\_cycle} \ , \tag{7.1}$$

where the number of blocks per cycle is 1 for a fully unrolled cipher, and greater than 1 for any design that re-uses the round structures to process a single input. The latency required to encrypt a single block was calculated as:

$$Lat = \frac{10 \cdot stages\_per\_round}{f_{clk}} \ , \tag{7.2}$$

where the number of clock cycles needed to process a single round is an average and may be a non-integer value. The estimated area efficiency of each implementation was calculated by the following metric:

$$Eff = \frac{Tput}{N_{slice}} \ , \tag{7.3}$$

which is measured in throughput rate (bps) per utilized CLB slice. As an alternative method of measuring efficiency, the following metric was used to estimate the time required to perform a common task:

$$T_{Enc} = Lat + \frac{num\_bits - 128 \cdot rep\_factor}{Tput} \ . \tag{7.4}$$

Assuming a constant stream of num bits of data, this equation captures the effects of latency and pipeline throughput on efficiency. The replication factor also plays a role in the metric, as the cipher will concurrently process multiple plaintext inputs when operating at maximum efficiency. For each design, we estimated the time to encrypt 1.5 KB of data, which is a commonly-observed payload size for Ethernet frames.

7.4.1. Stand-Alone Results

A selection of our experimental results can be found in Table 7.1. Each design is labeled R*W*-U*X*-P*YZ*, where the integer W corresponds to the cipher replication factor and X corresponds to the round unrolling factor. The Y value specifies the amount of transformation partitioning and pipelining: for $Y = 0$ the design has no pipelining; for $Y = 1$ each unrolled round is pipelined; for $Y = 2$ each round is split into two stages; and for $Y = 3$ each round is split into three stages. Finally, the Z value specifies the S-box technology

mapping in the design, where Z = [b] corresponds to BRAM, Z = [d] corresponds to distributed ROM primitives, and Z = [l] corresponds to logic gates. For the sake of clarity and brevity, unexceptional results from the set of possible designs were pruned when forming Table 7.1.

The first noteworthy trend that can be observed from these results is that cipher replication, while somewhat effective at improving performance, does not scale very well. Consider, for instance, the `R*-U5-P3b` design family. Slice usage increases at the same rate as the replication factor, but the additional stress of routing this increased number of slices drives the clock rate down. The resulting throughput improves by a factor of 1.77 between `R1-U5-P3b` and `R2-U5-P3b`, then by 1.25 from `R2-U5-P3b` to `R3-U5-P3b`. This less than linear improvement manifests itself in a 26% reduction in area efficiency from `R1-U5-P3b` to `R3-U5-P3b`. However, these highly-replicated designs are the fastest at the 1.5 KB encryption metric, due to their heavy I/O resource usage.

As expected, unrolling also leads to a dramatic increase in reconfigurable resource usage. `R2-U10-P1d`, for instance, uses over 8 times the number of slices than does `R2-U1-P1d`. Throughput improves at a close pace when unrolled rounds are pipelined, as pipelining allows a greater number of plaintext blocks to be simultaneously processed at maximum throughput.

Partitioning and pipelining of the individual transformations is effective at reducing the critical path and increasing performance. For example, `R3-U1-P3d` is able to run at over 2 times the clock frequency of `R3-U1-P0d`. While the faster design does not require very many additional slices, it does suffer an increase in latency. Overall the impact is a

positive one on efficiency; for example the `R1-U2-P3b` design slightly edges `R1-U2-P0b` in terms of Mbps/slice and is 1.71× as fast at running the 1.5 KB encryption test.

For several designs, distributed ROM primitives results in slightly higher clock rates than BRAM. The advantage of using BRAM for S-box synthesis can be seen in both area and efficiency. For instance, `R2-U5-P3b` saves 67.82% of the slices that `R2-U5-P3d` requires. Since the performance of the two designs is similar, this choice leads to a fourfold increase in area efficiency. It is important to note, however, that this seemingly impressive effect is primarily due to the inability of the efficiency metric to capture the area contribution of the BRAM modules. Using logic gates to directly implement the S-box transformations proves sub-optimal in all measured metrics when compared to using distributed ROM or BRAM.

Several of our designs do not fit on the XC2V8000 FPGA due to resource restrictions. The primary limitation on the allowable cipher replication factors is the availability of on-chip I/O. Since we do not implement any data serialization capabilities, incrementing the cipher replication factor requires an additional 256 IOBs. Given the fixed amount of 1102 IOBs on the target device, any design with $W > 3$ cannot be implemented. BRAM availability also restricts which designs can be implemented on the XC2V8000. For example, the `R2-U10-P*b` and `R3-U10-P*b` families require 200 and 300 Block SelectRAMs respectively, which exceeds the 168 available on the target device. Lastly, a few of our designs, such as the `R3-UF10-P*1` family, require more than the 46592 available slices.

The bolded values in Table 1 represent the designs which are maximal in terms of the measured area, performance, and efficiency characteristics. As expected, the designs requiring the least number of slices are those that utilize BRAM modules and have minimal

replicating, unrolling, and pipelining. The most aggressively partitioned and pipelined designs achieve the highest clock speeds, and the `R1-U10-P3d` is overall the fastest at 180.73 MHz. That same cipher design replicated twice (`R2-U10-P3d`) achieves the fastest theoretical throughput, at 31.71 Gbps. It is important to note that only by exploiting both the fine-grained and coarse-grained parallelism are we able achieve a maximum throughput greater than 30 Gbps. The `R1-U2-P0b` design demonstrates the usefulness of partial unrolling, as it has the lowest latency of all designs at 59.08 ns. The `R1-U10-P2b` design has the highest area efficiency (3.98 Mbps/slice), while an aggressively replicated design `R3-U5-P3d` is capable of encrypting a 1.5 KB block the quickest (0.71 μs).

7.4.2. Coprocessor Implementation

As mentioned in the previous chapter, the Xilinx ML310 is a Virtex-II Pro-based embedded development platform. It includes an XC2VP30 FPGA with two embedded PowerPC processors, DDR memory, PCI slots, ethernet, and standard I/O on an ATX board [**96**]. It functions much like a standard PC, with the addition of reconfigurable logic offered by the FPGA. In order to evaluate the performance of our standalone AES design as a cryptographic coprocesssor, the implementation is modified such that it is capable of communicating with the processor and main memory (Fig. 7.6). The "PLB I/F" component provides the AESencrypt module with an interface to the Processor Local Bus (PLB), which is the bus connecting the embedded processor to main memory and other high-speed peripherals. Through this bus interface, the processor can instruct the AESencrypt module to begin encrypting a stream of data from main memory. Additionally,

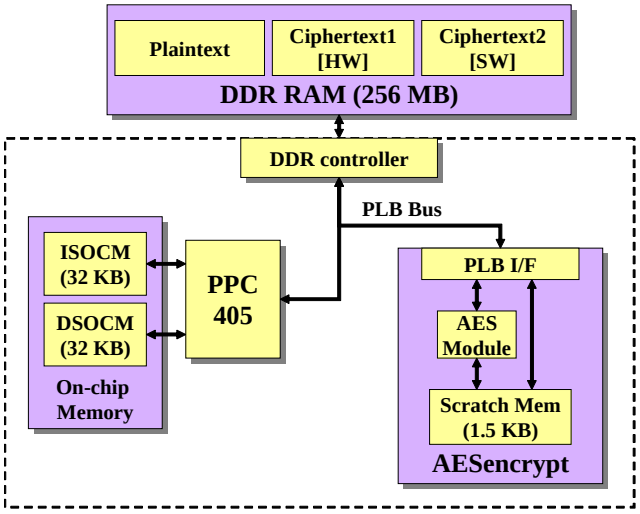

Figure 7.6. AES on ML310

the AESencrypt module is able to raise an interrupt when it has finished encrypting the data block.

This coprocessor implementation of our AES design is capable of operating in 3 modes: mode 1, the software benchmark, transfers plaintext data to the processor, performs the encryption in software, then transfers the encrypted data back to memory; mode 2, the first hardware mode, transfers plaintext to the AESencrypt module for encryption and stores the ciphertext to scratch memory within AESencrypt module; mode 3, the second hardware mode, performs the steps taken in mode 2 before transferring the stored ciphertext back to main memory.

The `R1-U10-P2b` design from the previous section was converted to a PLB peripheral and synthesized using the Xilinx ISE 7 toolchain. The design was configured to run at 100 MHz, the speed of the PLB, and software timers were used to measure the time required to encrypt 1.5 KB of data. The software benchmark mode required 2160 μs to fetch the plaintext from main memory, encrypt it, and write the ciphertext results back

to memory. The first hardware mode required 11 μs to transfer plaintext from memory to the AESencrypt module, encrypt it, and store it to the scratch memory on the module. The second hardware mode required 19 μs to perform the steps required by mode 2 in addition to writing the ciphertext back to main memory.

The most promising observation that can be made from these results is that the hardware modes are much faster than the software benchmark. Even the slower of the two hardware modes, for instance, is faster than the software benchmark by a factor of 113. This observed speedup highlights the potential of customized hardware for accelerating the performance of cryptographic algorithms over equivalent software implementations.

Comparing the hardware mode results to the stand-alone implementation of `R1-U10-P2b` (Table 7.1) is somewhat less encouraging. Under ideal conditions, the stand-alone implementation is capable of encrypting a 1.5 KB block of data in .98 μs, roughly 5% of the time required by the slower of the two hardware coprocessor modes. This performance gap can be attributed to a few factors. First, the hardware coprocessor operates at the speed of the PLB (100 MHz) instead of the maximum 117 MHz of the stand-alone design. Second, the 64-bit data width of the PLB prevents the AESencrypt module from processing plaintext blocks at maximum speed. Since each plaintext block must be 128-bits, the AESencrypt module effectively stalls on every other clock cycle while it waits for enough data to be brought across the PLB. Third, the coprocessor system incurs a PLB transfer overhead associated with mastering the bus and configuring the peripherals for transfer. Lastly, the coprocessor system incurs large interrupt overheads. For the hardware modes, the AESencrypt raises two interrupts: one to inform the processor that it is ready to

accept data from main memory; and one to inform the processor that the encryption process is complete.

7.4.3. Related Results

As previously discussed, there exists a wealth of recent academic and commercial FPGA implementations of cryptographic algorithms. Direct comparisons between implementations are somewhat difficult, due to the differences in the FPGAs that they target; however, the recent standardization of AES has given rise to many Xilinx-targeted implementations which can be used as a basis for comparison.

For instance, Helion Technology reports a maximum throughput exceeding 16 Gbps [**37**] for its high-performance commercial AES core. Several academic groups have reported high throughput values for designs that are similar to our most aggressively pipelined version. Standaert et al. [**77**] created a cipher that operated at over 14 Gbps. An AES design by Jarvinen et al. [**47**] performed at 17.8 Gbps. Finally, Saggese et al. [**71**] reached just over 20 Gbps. Other high-performance AES implementations include those of Hodjat and Verbauwhede [**39**], who designed a 21.54 Gbps core. Using similar approaches to these, we were previously able to design a 23.57 Gbps core [**103**], while the parallel AES architecture presented by Swankoski et al. [**78**] obtained a maximum throughput of 18.8 Gbps. Our superior throughput numbers can be explained by two factors: first, no previous approach in an academic or commercial setting has exploited multiple granularities of parallelism in a single design; second, by doubly packing S-boxes into Block SelectRAMs, we were able to generate a completely unrolled and highly replicated design that fit into our target device.

As a comparison to non-FPGA technologies, a hand-optimized assembly implementation of AES encryption in feedback mode achieved 1.538 Gbps on a 3.2 MHz Pentium IV processor [**56**]. An ASIC version of the design targeting 0.18 μm technology was able to achieve between 30 and 70 Gbps [**38**]. The fact that an ASIC implementation outperforms (in terms of throughput) any published FPGA implementation is not surprising given the reconfigurable overhead of FPGAs. Considering that high-throughput FPGA designs are quite competitive with their ASIC counterparts, this shortcoming becomes tolerable when considering the advantages to reconfigurable technology previously discussed.

Table 7.1. AES-128E implementation results for a Xilinx XC2V8000 FPGA

Design Name	Resource Usage		Performance			Efficiency	
	N_{slice}	N_{bram}	f_{clk} (MHz)	$Tput$ (Gbps)	Lat (ns)	($\frac{\text{Mbps}}{\text{slice}}$)	E[1.5KB] (μs)
R1-U1-P0d	1786	**0**	75.34	0.96	132.74	0.54	12.74
R1-U1-P1b	**485**	10	108.38	1.39	92.27	2.86	8.86
R1-U1-P2l	2801	0	101.09	1.29	197.84	0.46	9.60
R1-U1-P3d	1943	0	148.08	1.90	202.59	0.98	6.62
R1-U2-P0b	764	20	84.63	2.17	**59.08**	2.84	5.67
R1-U2-P1d	3324	0	68.86	1.76	145.22	0.53	7.04
R1-U2-P2l	5014	0	72.10	1.85	277.40	0.37	6.87
R1-U2-P3b	1267	20	151.91	3.89	197.49	3.07	3.32
R1-U5-P1l	12963	0	34.04	2.18	293.75	0.17	5.87
R1-U5-P2b	2213	50	137.04	8.77	145.94	3.96	1.53
R1-U5-P3b	2837	50	147.54	9.44	203.34	3.33	1.49
R1-U10-P1b	2515	100	75.09	9.61	133.17	3.82	1.40
R1-U10-P2b	3765	100	117.21	15.00	170.64	**3.98**	0.98
R1-U10-P3d	16941	0	**180.73**	23.13	165.99	1.37	0.69
R1-U10-P3l	29292	0	57.46	7.36	522.09	0.25	2.18
R2-U1-P0l	5788	0	50.80	1.30	196.86	0.22	9.45
R2-U1-P1d	3537	0	84.76	2.17	117.98	0.61	5.66
R2-U1-P2d	3659	0	111.21	2.85	179.84	0.78	4.41
R2-U1-P3b	1484	20	145.18	3.72	206.64	2.50	3.44
R2-U2-P0b	1476	40	73.15	3.75	68.35	2.54	3.28
R2-U2-P2b	2026	40	138.64	7.10	144.26	3.50	1.84
R2-U2-P3d	7330	0	116.10	5.94	258.39	0.81	2.28
R2-U2-P3l	11408	0	95.16	4.87	315.27	0.43	2.78
R2-U5-P1d	15960	0	47.45	6.07	210.75	0.38	2.19
R2-U5-P2b	4416	100	117.36	15.02	170.42	3.40	0.97
R2-U5-P2l	29103	0	61.98	7.93	322.70	0.27	1.84
R2-U5-P3b	5692	100	130.21	16.67	230.40	2.93	0.95
R2-U5-P3d	17687	0	97.53	12.48	307.59	0.71	1.27
R2-U10-P1d	30528	0	59.20	15.16	168.92	0.50	0.96
R2-U10-P2d	31694	0	65.64	16.80	304.70	0.53	1.02
R2-U10-P3d	33757	0	123.89	**31.71**	242.16	0.94	0.62
R3-U1-P0d	5243	0	66.08	2.54	151.34	0.48	4.84
R3-U1-P1l	8418	0	54.45	2.09	183.65	0.25	5.88
R3-U1-P2b	1843	30	124.91	4.80	160.12	2.60	2.64
R3-U1-P3b	2230	30	137.89	5.30	217.56	2.37	2.47
R3-U1-P3d	5831	0	156.91	6.03	191.19	1.03	2.17
R3-U2-P0b	2179	60	67.20	5.16	74.40	2.37	2.38
R3-U2-P1b	2271	60	85.54	6.57	116.90	2.89	1.93
R3-U2-P1l	16001	0	54.39	4.18	183.86	0.26	3.03
R3-U2-P2b	3032	60	124.92	9.59	160.10	3.16	1.40
R3-U2-P2d	10340	0	108.81	8.36	183.81	0.81	1.61
R3-U5-P1b	4751	150	69.48	13.34	143.93	2.81	1.04
R3-U5-P2b	6635	150	97.21	18.66	205.74	2.81	0.84
R3-U5-P2l	43750	0	68.60	13.17	291.56	0.30	1.20
R3-U5-P3b	8513	150	108.73	20.88	275.91	2.45	0.85
R3-U5-P3d	26543	0	129.63	24.89	231.42	0.94	**0.71**
R3-U10-P1d	45818	0	46.71	17.94	214.09	0.39	0.88

CHAPTER 8

Conclusions and Future Work

Both the methods used by hackers in compromising a software system and the preventative techniques developed in response thereto have received an increased level of attention in recent years, as the economic impact of both piracy and malicious attacks continues to skyrocket. Many of the recent approaches in the field of software protection have been found wanting in that they either a) do not provide enough security, b) are prohibitively slow, or c) are not applicable to currently available technology. This paper describes a novel architecture for software protection that combines a standard processor with dynamic tamper-checking techniques implemented in reconfigurable hardware. We have evaluated two distinct examples that demonstrate how such an approach provides much flexibility in managing the security/performance tradeoff on a per-application basis. Our results show that a reasonable level of security can be obtained through a combined obfuscating and tamper-proofing technique with less than a 20% performance degradation for most applications. When a highly-transparent solution is desired, FPGA resources not allocated for software protection can be used to mask some of the high latency operations associated with symmetric block ciphers.

Future work on this project will include implementing a method for calculating and storing hash values on the FPGA in order to secure data and file storage. Also, while our current simulation infrastructure is adequate for the experiments presented in this paper, it is also inherently limited in the sense that the component delays are completely

user-defined. Although we intelligently estimate these values based on previous work in hardware design, in the future it would be considerably more convincing to assemble the results based on an actual synthesized output. This would require our compiler to be modified to generate designs in either a full hardware description language such as VHDL or Verilog, or at a minimum, a specification language that can drive pre-defined HDL modules. Taking this concept a step further, as our techniques can be fully implemented with commercial off-the-shelf components, it would be useful to port our architecture to an actual hardware board containing both a processor and FPGA. Such a system would allow us to obtain real-world performance results and also to refine our threat model based on the strengths and weaknesses of the hardware.

8.1. A Secure Execution Environment using Untrusted Foundries

As the cost of manufacturing microprocessors continues to rise, chip makers have begun accelerating the export of fabrication capabilities to overseas locations. This trend may one day result in an adversarial relationship between system builders and chip producers. In this scenario the foundries may hide additional circuitry to not only compromise operation or leak keys, but also to enable software-triggered attacks and to facilitate reverse engineering. At the same time, users desire modern computing platforms with current software, along with the ability to run both secure and non-secure applications at will. In this section we describe an approach to address this *untrusted foundry* problem. We demonstrate a future method by which system builders can design trusted systems using untrusted chips from these remote foundries. The key principle behind our approach

is that we use a multi-layer encryption protocol alongside a dual-processor architecture. Using this approach any information leaked to an attacker will be of little value.

8.1.1. Introduction

Computing systems today are manufactured in a complex marketplace that features multiple chains of consumers and producers. In this marketplace, for example, a hardware producer such as Motorola sells chips but is itself a consumer when it contracts out the actual physical production of its design to a foundry. Similarly, producers of end-user devices such as cellphones are consumers of the chips that go into their devices. A recent development in this marketplace is that as the chip technologies pursue ever smaller dimensions, foundries have become extremely expensive to build and maintain [**45**]. As a result, the economics of this marketplace have led to the current situation where most foundries are concentrated outside the United States and Europe. This situation raises the possibility that such external foundries, which are not subject to the tight security customary in government facilities, may be compromised. In the case of a microprocessor, the most harmful type of compromise arises when an adversary exploits the sheer complexity of modern circuits to insert a small, hard-to-detect, so-called Trojan circuit into the chip.

What advantage might an adversary secure with such a hidden Trojan circuit? We describe possible attacks on end products such as radios or computers based on such chips even when the software has been subject to rigorous standards of security in its design and operation. In the case that the end product is a computer, such a high standard usually means the entire execution occurs in encrypted form [**54**] - software is completely

encrypted in main memory and is only decrypted at the time of execution in the processor. A hidden circuit inside the processor can easily thwart this approach by seizing control of the processor at an opportune moment and writing out decryption keys onto peripherals. This attack is commonly called a leakage attack. An even more rudimentary attack is to simply halt the processor at the worst possible moment, at a critical or random time long after the processor has been in use. Hidden circuits can also be set up to scan for electromagnetic signals so that a processor can be shutdown when provided the right external cue. Another useful attack from the point of view of the adversary is to use such circuits to facilitate reverse engineering of system design. In this sense, the problem we have described is not simply limited to military or government applications but is also of interest to commercial entities concerned with guarding their intellectual property.

This work focuses on the design of systems using chips from untrusted foundries. We do not consider the equally important problem of detecting such circuits by subjecting chips to external black-box tests or imaging techniques. Our primary goal is to detect the moment such a hidden circuit "makes its move" and then to raise an alarm to the user. We assume that the system is itself built in a trusted location and that no collusion exists between foundries and system developers (such as board makers or software developers). Furthermore, our solution is targeted at a very high level of security and therefore we assume encrypted execution, as will be explained in the next section. Our approach to the untrusted foundry problem can be summarized as follows. We utilize a multi-layer encryption protocol, a two-processor architecture, and a trusted tool-chain under the supervision of architects and developers of the targeted secure applications. The core of

our approach is the way in which the two processors are used: one is configured as a gateway, while the other is configured as an execution engine.

8.1.2. Related Work

Many previous secure architectures have focused on providing an encrypted execution environment. Figure 8.1 shows an encrypted program residing in main memory where the cryptographic keys are in the processor. When instructions or data are loaded into the processor, they are decrypted inside the processor. Likewise data that is written back to main memory is encrypted using these same stored keys. The purpose of the encryption is to prevent information leakage over the bus. Such leakage can occur when the product is captured by an attacker with access to a sophisticate laboratory in which the attack can snoop on the bus or directly manipulate the bus itself.

Encrypted execution has been studied by various researchers. In [**54**], an architecture for tamper-resistant software is proposed, based on an eXecute-Only Memory (XOM) model that allows instructions stored in memory to be executed but not manipulated. Specialized hardware is used to accelerate cryptographic functionality needed to protect data and instructions on a per-process basis. Much of the recent research in this field has focused on improving the overhead of the required cryptographic computation in the memory fetch path - examples include the architectural optimizations proposed in [**74, 79, 99**].

Pande et al. [**104, 105**] address the problem of information leakage on the address bus wherein the attacker would be snooping the address values to gain information about the control flow of the program. They provide a hardware obfuscation technique which

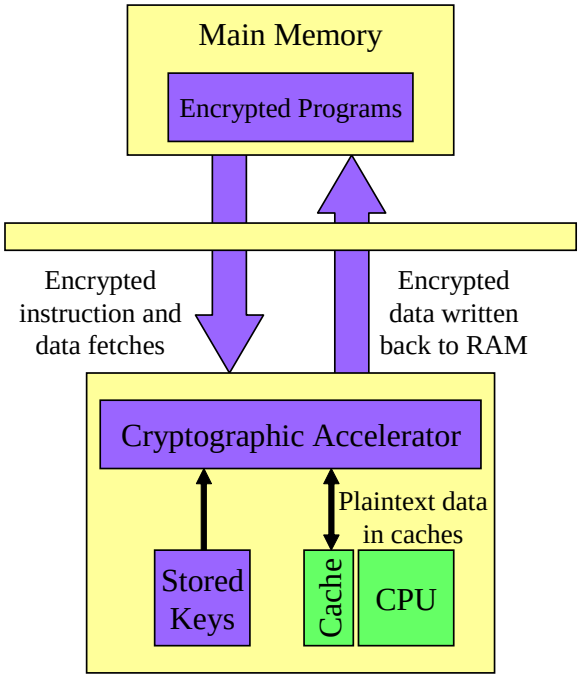

Figure 8.1. Encrypted execution and data platforms

is based on dynamically randomizing the instruction addresses. This is achieved through a secure hardware coprocessor which randomizes the addresses of the instruction blocks, and rewrites them into new locations.

Some of the earliest work in this area involves a number of secure coprocessing solutions. Programs, or parts of the program, can be run in an encrypted form on these devices thus never revealing the code in the untrusted memory and thereby providing a tamper resistant execution environment for that portion of the code. A number of secure coprocessing solutions have been designed and proposed, including systems such as IBM's Citadel [**90**], Dyad [**81**], and the Abyss [**89**].

Smart cards can also be viewed as type of secure coprocessor; a number of studies have analyzed the use of smart cards for secure applications [**33, 52, 72**]. Sensitive

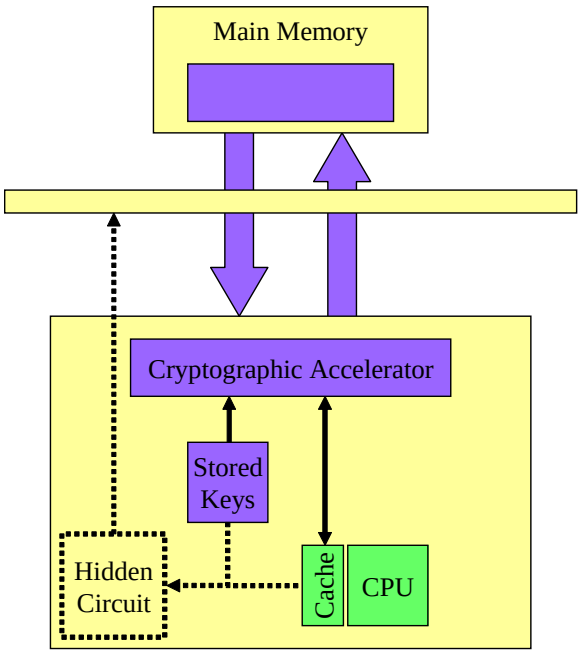

Figure 8.2. A hidden circuit leaking information

computations and data can be stored in the smart card but they offer no direct I/O to the user. Most smart card applications focus on the secure storage of data although studies have been conducted on using smart cards to secure an operating system [**17**]. As noted in [**15**], smart cards can only be used to protect small fragments of code and data.

8.1.3. Our Approach

Given this background into these encrypted execution and data systems, one observes that a hidden circuit in the processor can easily leak the keys onto the bus to aid an attacker, as shown in Figure 8.2.

Our approach includes features at both the architectural and compiler levels. These features can be incorporated into a standard architecture that also runs applications in non-secure mode for the sake of efficiency. Figure 8.3 shows this dual-use architecture.

The regular or non-secure applications execute as usual on a standard CPU, even though the data path from memory to CPU goes through the secure gateway processor (which we describe below). Standard applications are assumed to be constructed using a standard compiler tool chain. The box to the right shows how the same architecture can be applied to peripherals, a matter which we discuss in the following section.

The secure applications are doubly encrypted by the compiler and execute using the dual-processor components of this architecture. One of these processors serves as a gateway whose only function is to perform an initial decryption of the instructions and data before sending the results of this first-level decryption onto the execution processor. The execution processor decrypts this stream a second time to reveal the actual instructions. This part can be performed in a manner similar to the conventional encrypted execution depicted in Figure 8.1.

Data values that need to be written back to memory are encrypted first by the execution processor and then once again by the gateway using its own keys. Note that the execution processor is physically unable to access the bus; indeed the output of the execution processor is captured by the gateway processor so that any potential leakage can be examined at the gateway.

We assume that all components are individually assumed to be produced at untrusted foundries. Furthermore, we assume that no collusion exists between the foundries that produced the gateway processor and the one that produced the engine processor. We will examine the implications of this assumption in the following section.

We now describe to what extent our approach addresses the foundry threat. A foundry compromise of the execution processor can result in one of three problems:

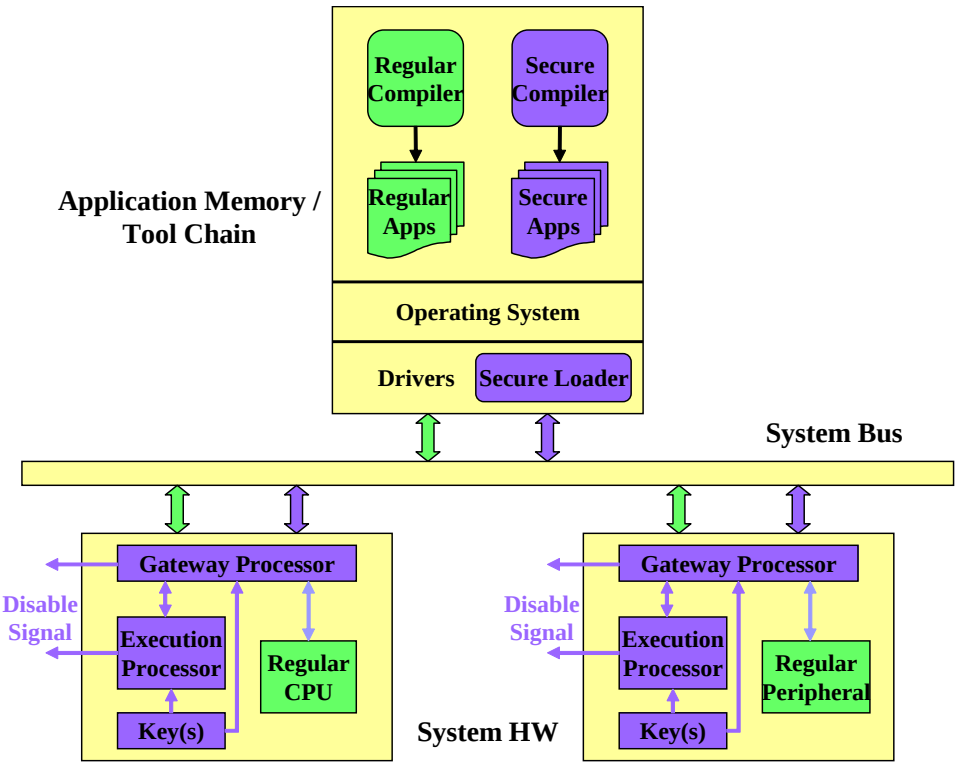

Figure 8.3. Our approach to obtaining security through untrusted foundries

(1) The execution processor can try to write to memory (either to leak information or to disrupt program flow)

(2) The execution processor can try to leak information on its pins

(3) The execution processor can simply stall or deny service

For the first problem, note that only the gateway is physically connected to memory (the execution processor is physically isolated); the gateway validates all writes to memory. For the second problem, the encryption mechanism at the gateway will detect invalid data. For the third problem, we can use a heartbeat (separately clocked timer) at the gateway to detect denial of service from the execution processor.

Now consider a foundry compromise of the gateway processor. A compromised gateway exposes different system vulnerabilities:

(1) The gateway will be able to leak information on the bus

(2) The gateway will be able to write anywhere in memory, including where secure applications reside

(3) The gateway will be able to deny service to the execution processor

For the first problem, observe that the gateway can only leak encrypted (by the execution processor) instructions, leaving the instructions secure. For the second problem, note that a write into secure memory can be eventually detected by the execution processor because the gateway is not provided access to the keys for the inner layer of encryption; therefore any overwriting of instructions will be detected as an improper instruction block by the engine. For the third problem, the denial of service can be detected through a heartbeat assigned to the engine processor. Finally, observe that the gateway can attempt a replay attack on the code. This would have to be done by the gateway in a blind fashion since the instructions are encrypted. Such replays can be detected through compiler techniques aimed at detecting control-flow attacks [**50, 102**].

In sum, the only way for a foundry attack to work is for both chips (foundries) to collude. In the next section, we describe ways in which this risk can be minimized.

8.1.4. Discussion

Some of the assumptions above are fairly straightforward and constitute standard practice today: (1) system design and implementation given the untrusted chips can be performed in a secure location; (2) there is no collusion between system designers and foundries; (3)

encryption is easily performed as part of the compilation process; (4) the two-processor architecture itself imposes no requirement on the software beyond dual-encryption.

However, one might question some of the other assumptions. Of these, perhaps the most troublesome is our assumption that no collusion exists between the foundries that manufactured the two processors. This assumption can be strengthened in several ways, as we now describe. First, we can reduce the possibility of collusion by using two chips made by foundries in different countries. Second, we can use soft processors (with reconfigurable technology) for one or both processor chips so that a foundry has no means of guessing the purpose of the chip. A third technique is to use keys that are programmed post-foundry, using either reconfigurable hardware or by hand using an electromechanical device. When used in combination, these techniques can help minimize the risk of collusion.

Although this is mainly a concept work at this stage, we can make a few comments related to performance. First, a naive implementation of the two processor architecture may significantly slow performance. An example of a naive implementation would be to implement the cryptographic functions in software. Although straightforward, such an implementation will quickly become the bottleneck and is likely to result in an order of magnitude degradation of performance in applications that are cache-sensitive. A better architectural implementation would involve building custom hardware that pipelines the various units of computation (using pipelined implementations of AES [**47, 103**], for example) so that the processor is never starved of instructions. At the same time, because the data is encrypted when written to memory, there will always be a substantial performance hit when a cache writeback occurs. This penalty would also be incurred in a

standard encrypted execution environment. Proper pipelining and prefetching [**57, 70**] can help mitigate the effect of our additional encryption.

The system as described above thus far addresses monolithic self-contained applications. Care must be taken to ensure that an operating system can execute in encrypted mode. This is not a trivial task by any means, because of the sometime complex interaction between peripherals, the processor and the operating system. If sensitive data is entered via peripherals, then those chipsets may be just as easily compromised by a foundry. Thus, the architecture needs to be properly extended to peripheral components as well.

8.1.5. Implementation Options

The key principle in our approach combines three ideas: the use of bi-layer encryption, dual mutually-distrusting processors and trusted compilation of secure applications. Note that this principle can be applied not only to hardware (as described above) but to software as well. As an example of the latter, consider a system constructed out of COTS processors running virtual machines. Two virtual machines (whether on the same board, or connected through the Internet) connected in the above architecture can perform the same roles as the two processors, as shown in Figure 8.4.

There are several advantages to starting with virtual machines: (1) proof-of-concept can be more easily and visibly demonstrated; (2) virtual machines can be instrumented with hooks for red team benchmarking; (3) they can be designed to provide accurate performance estimates; (4) they are easy to modify and replicate, and (5) they are much cheaper than hardware. Each box in Figure 8.4 represents a virtual machine running in

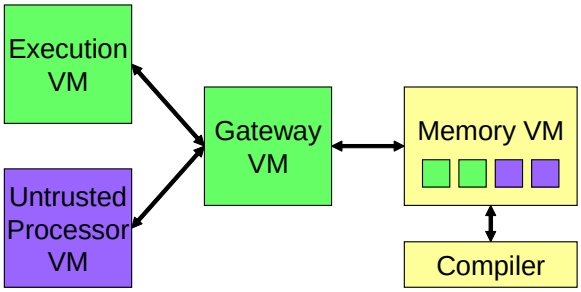

Figure 8.4. Virtual machine implementation

its own process. Two of the boxes represent virtual machines or emulators for the two processors in our secure component. The lower-left box represents the regular (untrusted) processor used for non-secure applications. Finally, on the right side, main memory is shown as containing both secure and non-secure applications. The secure applications are compiled by a trusted compiler that performs the bilayer encryption.

One interesting and valuable intermediate option between pure hardware and pure software is to use reconfigurable hardware built out of Field Programmable Gate Arrays (FPGAs). As previously-mentioned, FPGAs offer several advantages:

- They can be reconfigured in the field to support multiple instruction sets
- It is harder to hide hidden circuits in them because they are easier to test and their structure is simple
- Their technology curve scales as fast as ASIC technology
- They can be optimized both at the hardware and software levels through intelligent manipulation of design tradeoffs.

We plan on prototyping our hardware system using the Xilinx ML310 FPGA development board (Figure 6.3). As previously mentioned, the ML310 contains a XC2VP30 FPGA that has two PPC 405 processors and 13696 reconfigurable CLB slices. The board

and FPGA can act as a fully-fledged PC, with 256 MB on-board memory, some solid-state storage, and several standard peripherals.

The flexibility of the Virtex II Pro FPGA allows us to utilize the built-in PPC processors as our standard CPUs, while leveraging soft processors built using reconfigurable logic (such as the Xilinx MicroBlaze [**95**] processor) as our gateway processors.

References

[1] NIST Special Publication 800-38A. Recommendation for block cipher modes of operation. available at http://csrc.nist.gov/publications/, 2001.

[2] Actel Corporation. CoreDES data sheet, v2.0. available at http://www.actel.com, 2003.

[3] Actel Corporation. Design security with Actel FPGAs. available at http://www.actel.com, 2003.

[4] Mehboob Alam, Wael Badawy, and Graham Jullien. A novel pipelined threads architecture for AES encryption algorithm. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 296–302, 2002.

[5] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, pages 52–62, November 1999.

[6] ARM, Ltd. Application note 32: the ARMulator. available at http://www.arm.com, 1999.

[7] ARM, Ltd. ARM TrustZone technology. available at http://www.arm.com, 2004.

[8] David Aucsmith. Tamper-resistant software: An implementation. In *Proceedings of the 1st International Workshop on Information Hiding*, pages 317–333, May 1996.

[9] Dirk Baifanz, Drew Dean, and Mike Spreitzer. A security infrastructure for distributed Java applications. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 15–26, May 2000.

[10] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Stevn Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Proceedings of Advances in Cryptology (CRYPTO '01)*, pages 1–18, August 2001.

[11] Lujo Bauer, Michael A. Schneider, and Edward W. Felten. A proof-carrying authorization system. Technical Report CS-TR-638-01, Department of Computer Science, Princeton University, April 2001.

[12] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Message authentication using hash functions: the HMAC construction. *RSA Laboratories' CryptoBytes*, 2(1), Spring 1996.

[13] 'Bulba' and 'Kil3r'. Bypassing StackGuard and StackShield. *Phrack*, 10(56), May 2000.

[14] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, Department of Computer Science, University of Wisconsin-Madison, June 1997.

[15] Hoi Chang and Mikhail Atallah. Protecting software code by guards. In *Proceedings of the ACM Workshop on Security and Privacy in Digital Rights Management*, pages 160–175, November 2000.

[16] Spencer Chang, Paul Litva, and Alec Main. Trusting DRM software. In *Proceedings of the W3C Workshop on Digital Rights Management for the Web*, January 2001.

[17] Paul Clark and Lance Hoffman. BITS: A smartcard protected operating system. *Communications of the ACM*, 37(11):66–70, November 1994.

[18] Christian Collberg and Clark Thomborson. Software watermarking: models and dynamic embeddings. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 311–324, January 1999.

[19] Christian Collberg and Clark Thomborson. Watermarking, tamper-proofing, obfuscation: tools for software protection. *IEEE Transactions on Software Engineering*, 28(8):735–746, August 2002.

[20] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, The University of Auckland, July 1997.

[21] Christian Collberg, Clark Thomborson, and Douglas Low. Breaking abstractions and unstructuring data structures. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 28–38, May 1998.

[22] Computer Security Institute and Federal Bureau of Investigation. CSI/FBI 2002 computer crime and security survey. available at http://www.gocsi.com, April 2002.

[23] Mark Corliss, E. Lewis, and Amir Roth. Using DISE to protect return addresses from attack. In *Proceedings of the Workshop on Architectural Support for Security and Anti-Virus (WASSA)*, October 2004.

[24] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, January 1999.

[25] Joan Daeman and Vincent Rijmen. The block cipher Rijndael. In Jean-Jacques Quisquater and Bruce Schneier, editors, *Smart Card Research and Applications*, volume 1820 of *Lecture Notes in Computer Science*, pages 288–296. Springer-Verlag, 2000.

[26] Dallas Semiconductor. Features, advantages, and benefits of button-based security. available at http://www.ibutton.com, 1999.

[27] Andreas Dandalis, Viktor Prasanna, and José Rolim. An adaptive cryptographic engine for IPSec architectures. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 132–144, April 2000.

[28] Adam Elbirt, Wai Yip, B. Chetwynd, and Christof Paar. An FPGA implementation and performance evaluation of the AES block cipher candidate algorithm finalists. In *The Third Advanced Encryption Standard (AES3) Candidate Conference*, pages 13–27, April 2000.

[29] M. Ernst, Michael Jung, Felix Madlener, S. Huss, and Rainer Blumel. A reconfigurable system on chip implementation for elliptic curve cryptography over GF(2n). In *Proceedings of the 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 381–399, 2002.

[30] FIPS PUB 197. Advanced Encryption Standard (AES), National Institute of Standards and Technology, U.S. Department of Commerce. available at http://csrc.nist.gov, November 2001.

[31] FIPS PUB 46-3. Data Encryption Standard (DES), National Institute of Standards and Technology, U.S. Department of Commerce. available at http://csrc.nist.gov, 1999.

[32] Matt Fisher. Protecting binary executables. *Embedded Systems Programming*, 13(2), February 2000.

[33] Howard Gobioff, Sean Smith, Doug Tygar, and Bennet Yee. Smart cards in hostile environments. In *Proceedings of the 2nd USENIX Workshop on Electronic Commerce*, pages 23–28, November 1996.

[34] Ivan Gonzalez, Sergio Lopez-Buedo, Francisco J. Gomez, and Javier Martinez. Using partial reconfiguration in cryptographic applications: An implementation of the IDEA algorithm. In *Proceedings of the 13th International Conference on Field-Programmable Logic and its Applications (FPL)*, pages 194–203, 2003.

[35] Tim Grembowski, Roar Lien, Kris Gaj, Nghi Nguyen, Peter Bellows, Jaroslav Flidr, Tom Lehman, and Brian Schott. Comparative analysis of the hardware implementations of hash functions SHA-1 and SHA-512. In *Proceedings of the 5th International Conference on Information Security (ISC)*, pages 75–89, 2002.

[36] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th IEEE Annual Workshop on Workload Characterization*, pages 3–14, December 2001.

[37] Helion Technology. AES Xilinx FPGA core data sheet, 2003.

[38] Alireza Hodjat and Ingrid Verbauwhede. Minimum area cost for a 30 to 70 Gbits/s AES processor. In *Proccedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2003)*, pages 83–88, 2003.

[39] Alireza Hodjat and Ingrid Verbauwhede. A 21.54 Gbits/s fully pipelined AES processor on FPGA. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2004.

[40] Andreas Hoffmann, Heinrich Meyr, and Rainer Leupers. *Architecture Exploration for Embedded Processors using LISA*. Kluwer Academic Publishers, Boston, MA, 2002.

[41] Bill Horne, Lesley Matheson, Casey Sheehan, and Robert Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *ACM Workshop on Security and Privacy in Digital Rights Management*, pages 141–159, November 2001.

[42] IBM. Secure systems and smart cards. available at http://www.research.ibm.com/secure_systems, 2002.

[43] Intel. Intel LaGrande technology. available at http://www.intel.com, 2004.

[44] International Plannning and Research Corporation. Eighth annual BSA global software piracy study. available at http://www.bsa.org, June 2003.

[45] International Technology Roadmap for Semiconductors (ITRS). International technology roadmap for semiconductors, 2004 update. available at http://public.itrs.net, September 2005.

[46] Naomaru Itoi. Secure coprocessor integration with Kerberos V5. Technical Report RC-21797, IBM Research Division, T.J Waston Research Center, July 2000.

[47] Kimmo U. Jarvinen, Matti T. Tommiska, and Jorma O. Skytta. A fully pipelined memoryless 17.8 Gbps AES-128 encryptor. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 207–215, February 2003.

[48] Neil Johnson and Stefan Katzenbeisser. *A survey of steganographic techniques*. Artech House, 1999.

[49] Jens-Peter Kaps and Christof Paar. Fast DES implementation for FPGAs and its application to a universal key-search machine. In *Proceedings of the 5th Annual Workshop on Selected Areas in Cryptography (SAC)*, pages 234–247, 1998.

[50] Darko Kirovski, Milenko Drinic, and Miodrag Potkonjak. Enabling trusted software integrity. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, October 2002.

[51] Paul Kocher. Cryptanalysis of Diffie-Hellman, RSA, DSS and other systems using timing attacks. Extended abstract, December 1995.

[52] Oliver Kommerling and Marcus Kuhn. Design principles for tamper-resistant smartcard processors. In *Proceedings of the USENIX Workshop on Smartcard Technology*, pages 9–20, May 1999.

[53] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, pages 330–335, December 1997.

[54] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and M Horowitz. Architectural support for copy and tamper resistant

software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 168–177, November 2000.

[55] Roar Lien, Tim Grembowski, and Kris Gaj. A 1 Gbit/s partially unrolled architecture of hash functions SHA-1 and SHA-512. In *Proceedings of the Cryptographers' Track at the RSA Conference (CT-RSA)*, pages 324–338, 2004.

[56] Helger Lipmaa. AES Implementation Speed Comparison. available at http://www.tcs.hut.fi/˜helge/aes/rijndael.html, 2003.

[57] Chi-Keung Luk and Todd Mowry. Cooperative prefetching: Compiler and hardware support fo effective instruction prefetching in modern processors. In *Proceedings of the 31st Annual Internation Symposium on Microarchitecture (MICRO-31)*, December 1998.

[58] Wenbo Mao. *Modern Cryptography: Theory and Practice*. Prentice Hall, Upper Saddle River, NJ, 2004.

[59] Nele Mentens, Siddika Berna Ors, and Bart Preneel. An FPGA implementation of an elliptic curve processor over GF(2m). In *Proceedings of the 14th ACM Great Lakes Symposium on VLSI (GLVLSI)*, pages 454–457, 2004.

[60] Mentor Graphics. Modelsim SE simulation and verification datasheet. available at http://www.mentor.com, 2002.

[61] MIPS. SmartMIPS application-specific extension. available at http://www.mips.com, 2004.

[62] George Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languagers*, pages 106–119, January 1997.

[63] George Necula. web page http://www.cs.berkeley.edu/ñekula, 2003.

[64] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the 2nd USENIX Symposium on OS Design and Implementation*, pages 229–243, October 1996.

[65] Amaury Neve, Denis Flandre, and Jean-Jaques Quisquater. Feasibility of smart cards in Silicon-On-Insulator (SOI) technology. In *Proceedings of the USENIX Workshop on Smartcard Technology*, May 1999.

[66] Souichi Okada, Naoya Torii, Kouichi Itoh, and Masahiko Takenaka. Implementation of elliptic curve cryptographic coprocessor over GF(2m) on an FPGA. In *Proceedings of the 2nd International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 25–40, 2000.

[67] 'Aleph One'. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[68] Jonathan Pincus and Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4):20–27, July 2004.

[69] Viktor Prasanna and Andreas Dandalis. FPGA-based cryptography for internet security. In *Online Symposium for Electronic Engineers*, November 2000.

[70] Glenn Reinman, Brad Calder, and Todd Austin. Fetch directed instruction prefetching. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture (MICRO-32)*, November 1999.

[71] Giacinto Paolo Saggese, Antonio Mazzeo, Nicola Mazzoca, and Antonio G. M. Strollo. An FPGA-based performance analysis of the unrolling, tiling, and pipelining of the AES algorithm. In *Proceedings of the 13th International Conference on Field-Programmable Logic and its Applications (FPL)*, pages 292–302, 2003.

[72] Bruce Schneier and Adam Shostack. Breaking up is hard to do: modeling security threats for smart cards. In *Proceedings of the USENIX Workshop on Smartcard Technology*, pages 175–185, May 1999.

[73] Weidong Shi, Hsien-Hsin Lee, Mrinmoy Ghosh, Chenghuai Lu, and Alexandra Boldyreva. High efficiency counter mode security architecture via prediction and precomputation. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, pages 14–24, June 2005.

[74] Weidong Shi, Hsien-Hsin Lee, Mrinmoy Ghosh, Chenghuai Lu, and Alexandra Boldyreva. High efficiency counter mode security architecture via prediction and precomputation. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, pages 14–24, June 2005.

[75] Sean Smith. Secure coprocessing applications and research issues. Technical Report LA-UR-96-2805, Computer Research and Applications Group, Los Alamos National Laboratory, August 1996.

[76] Sean Smith and Steve Weingart. Building a high-performance programmable secure coprocessor. *Computer Networks*, 31(9):831–860, April 1999.

[77] Francois-Xavier Standaert, Gael Rouvroy, Jean-Jacques Quisquater, and Jean-Didier Legat. A methodology to implement block ciphers in reconfigurable hardware and its application to fast and compact AES Rijndael. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 216–224, February 2003.

[78] E.J. Swankoski, Richard R. Brooks, Vijaykrishnan Narayanan, Mahmut Kandemir, and Mary Jane Irwin. A parallel architecture for secure FPGA symmetric encryption. In *Proceedings of the Reconfigurable Architectures Workshop (RAW)*, 2004.

[79] Reed Taylor and Seth Goldstein. A high-performance flexible architecture for cryptography. In *Proceedings of the Workshop on Cryptographic Hardware and Software Systems*, August 1999.

[80] Trusted Computing Group. available at http://www.trustedcomputing.org, 2005.

[81] Doug Tygar and Bennet Yee. Dyad: A system for using physically secure coprocessors. Technical Report CMU-CS-91-140R, Department of Computer Scence, Carnegie Mellon University, May 1991.

[82] Doug Tygar and Bennet Yee. Dyad: A system for using physically secure coprocessors. In *Proceedings of the Harvard-MIT Workshop on Protection of Intellectural Property*, April 1993.

[83] Ramarathnam Venkatesan, Vijay Vazirana, and Saurabh Sinha. A graph theoretic approach to software watermarking. In *Proceedings of the Fourth International Information Hiding Workshop*, April 2001.

[84] VIA. Padlock hardware security suite. available at http://www.via.com, 2004.

[85] Chenxi Wang, Jack Davidson, Jonathan Hill, and John Knight. Protection of software-based survivability mechanisms. In *Proceedings of the 2001 IEEE/IFIP International Conference on Dependable Systems and Networks*, July 2001.

[86] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software tamper resistance: obstructing the static analysis of programs. Technical Report CS-2000-12, Department of Computer Science, University of Virginia, May 2000.

[87] Steve Weingart. Physical security for the mABYSS system. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 52–58, April 1987.

[88] Steve Weingart, Steve White, William Arnold, and Glenn Double. An evaluation system for the physical security of computing systems. In *Proceedings of the 6th Computer Security Applications Conference*, pages 232–243, December 1990.

[89] Steve White and Liam Comerford. ABYSS: A trusted architecture for software protection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 38–51, April 1987.

[90] Steve White, Steve Weingart, William Arnold, and Elaine Palmer. Introduction to the Citadel architecture: Security in physically exposed environments. Technical Report RC 16682, IBM Research Division, T.J Waston Research Center, May 1991.

[91] Thomas Wollinger and Christof Paar. How secure are FPGAs in cryptographic applications? In *Proceedings of the 13th International Conference on Field-Programmable Logic and its Applications (FPL)*, pages 91–100, 2003.

[92] Jeremy Wyant. Establishing security requirements for more effective and scalable DRM solutions. In *Proceedings of the Workshop on Digital Rights Management for the Web*, January 2001.

[93] Xilinx, Inc. High-speed data serialization and deserialization (840 Mb/s LVDS). available at http://www.xilinx.com, 2003.

[94] Xilinx, Inc. Virtex-II Pro Platform FPGA data sheet. available at http://www.xilinx.com, 2003.

[95] Xilinx, Inc. MicroBlaze processor reference guide. available at http://www.xilinx.com, 2005.

[96] Xilinx, Inc. ML310 user guide. available at http://www.xilinx.com, 2005.

[97] Xilinx, Inc. RocketIO tranceiver user guide. available at http://www.xilinx.com, 2005.

[98] Jun Yang, Youtao Zhang, and Lan Gao. Fast secure processor for inhibiting software piracy and tampering. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO)*, pages 351–360, December 2003.

[99] Jun Yang, Youtao Zhang, and Lan Gao. Fast secure processor for inhibiting software piracy and tampering. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO)*, pages 351–360, December 2003.

[100] Bennet Yee. Using secure coprocessors. Technical Report CMU-CS-94-149, Computer Science Department, Carnegie Mellon University, May 1994.

[101] Bennet Yee and Doug Tygar. Secure coprocessors in electronic commerce applications. In *Proceedings of the 1st USENIX Workshop on Electronic Commerce*, pages 155–170, July 1995.

[102] Joseph Zambreno, Alok Choudhary, Rahul Simha, and Bhagi Narahari. Flexible software protection using HW/SW codesign techniques. In *Proceedings of Design, Automation, and Test in Europe*, pages 636–641, February 2004.

[103] Joseph Zambreno, David Nguyen, and Alok Choudhary. Exploring area/delay tradeoffs in an AES FPGA implementation. In *Proceedings of the 14th International Conference on Field Progr ammable Logic and Applications (FPL)*, pages 575–585, 2004.

[104] Xiaotong Zhuang, Tao Zhang, Hsien-Hsin Lee, and Santosh Pande. Hardware assisted control flow obfuscation for embedded processors. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, October 2004.

[105] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2004.