High-Performance Software Protection Using Reconfigurable Architectures

JOSEPH ZAMBRENO, STUDENT MEMBER, IEEE, DAN HONBO, STUDENT MEMBER, IEEE, ALOK CHOUDHARY, FELLOW, IEEE, RAHUL SIMHA, MEMBER, IEEE, AND BHAGIRATH NARAHARI

Invited Paper

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. 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.

Keywords*—*Field programmable gate arrays (FPGAs), reconfigurable architectures, security, software protection.

I. Introduction and Motivation

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 [1]. 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 [2]. 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.

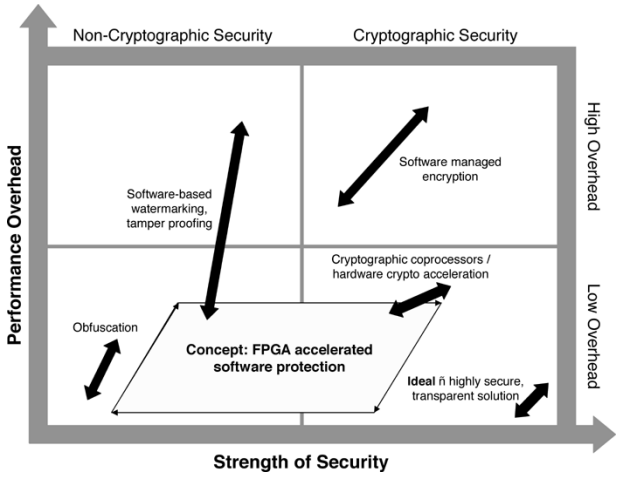


Fig. 1. Performance and security strength of various software protection approaches.

Manuscript received September 24, 2004; revised March 18, 2005. This work was supported in part by the National Science Foundation (NSF) under Grant CCR-0325207 and also by an National Science Foundation graduate research fellowship.

J. Zambreno, D. Honbo and A. Choudhary are with the Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208 USA (e-mail: zambro1@ece.northwestern.edu; d-honbo@northwestern.edu; choudhar@ece.northwestern.edu).

R. Simha and B. Narahari are with the Department of Computer Science, George Washington University, Washington, DC 20052 USA (e-mail: simha@gwu.edu; narahari@gwu.edu).

Digital Object Identifier 10.1109/JPROC.2005.862474

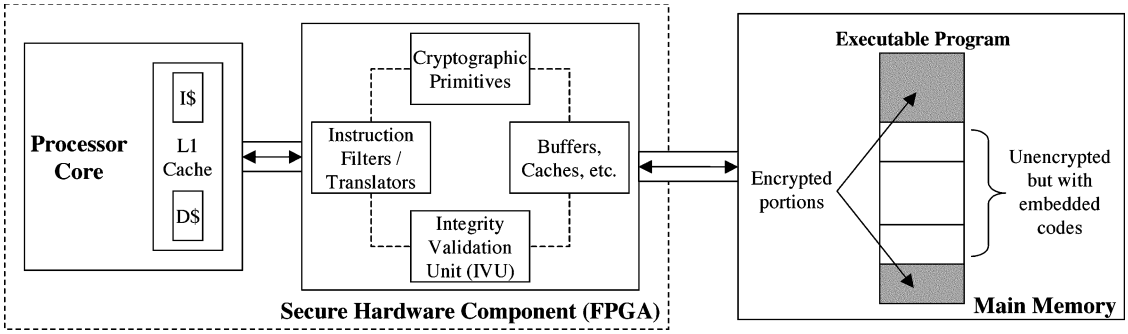


Fig. 2. Conceptual view.

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). 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 Section II.

Also, as can be inferred from Fig. 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. As can be seen in Fig. 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 runtime 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.

The remainder of this paper is organized as follows. In Section II we provide additional background into the field of software protection, with a review of some of the more commonly-used defensive techniques. Section III 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 Section IV we present our architecture in more detail, illustrating how we can utilize an FPGA situated in the instruction stream 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. Section V discusses the performance implications of our approach, first by explicitly quantifying the security/performance tradeoff and then by performing experimental analysis. Finally, in Section VI, we present our conclusions alongside a discussion of future techniques that are currently in development.

II. Background

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 [3] detailed below.

Watermarking is a technique whereby messages are hidden inside a piece of software in such a way that they can be reliably identified [4]. 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 structures can then be verified at runtime. Venkatesan *et al.* present an interesting variation on watermarking in [5], 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 runtime if it has been altered. A type of dynamic "self-checking" is proposed in both [6] and [7]. These approaches assert application integrity by essentially inserting instructions to perform code checksums during program execution. 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. One flaw with many of these tamper-proofing approaches is that in most architectures it is relatively easy to build an automated tool to reveal the checking mechanisms [9]. For example, checksum computations can be easily identified by finding code that operates directly on the instruction address space. Accordingly, the security of these approaches depends heavily on the security of the checking mechanisms themselves.

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 [10]. Safety rules, as part of a theorem-proving technique, are used on the host to guarantee proper program behavior. 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 [11]. Obfuscation techniques range from simple encoding of constants to more complex methods that completely restructure code while maintaining correctness. 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 [12].

A. *Other Hardware-Based Approaches*

Using our definition, there have been several hardware-based software protection approaches. **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 [13], Dyad [14], and the Abyss [15].

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 [16]. 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 [6], smart cards can only be used to protect small fragments of code and data.

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 [18] implements specialized processor instructions designed to accelerate software cryptography algorithms, whereas VIA's Padlock Hardware Security Suite [19] adds a full AES encryption engine to the processor die. Both extensions seek to eliminate the need for cryptographic coprocessors.

Intel's LaGrande Technology [20], ARM's TrustZone Technology [21], 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 [22]. 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.

B. Closely Related Work

In [23], 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 encrypted execution and data platforms.

In [24], 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 runtime 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.* [25] 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.

III. System Model and Rationale

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.

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 encrypted execution and data (EED) 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 subcontracted 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.

A. Focus Areas

The contributions of our work can be categorized into four areas as seen in Fig. 3. 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

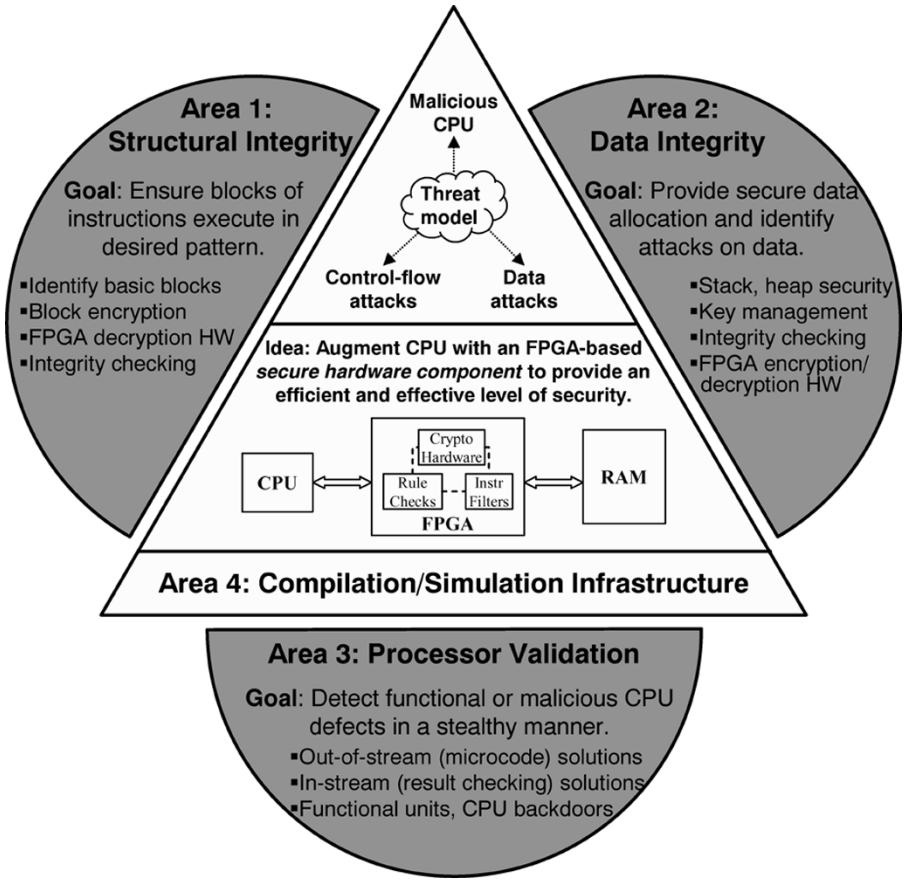


Fig. 3. Research focus areas as they relate to the conceptual view.

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 metainstructions 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).

B. Threat Model

As mentioned in Section I, 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.

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 can 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 reissue 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. 4.
- **Control-flow attacks**—the sophisticated attacker can elucidate the control-flow structure of a program

Addr	Instruction
0x00f0	ADD r12,r12,#0xe0
0x00f4	ADD r3,r13,#0xa0
0x00f8	SUB r12,r1,r0
0x00fc	ADD r14,r13,#4
0x0100	LDR r12,[r14,r12,LSL #2]
0x0104	LDR r3,[r3,r0,LSL #2]
0x0108	LDR r2,[r2,r1,LSL #2]
0x010c	ADD r0,r0,#1
0x0110	MLA r2,r12,r3,r2

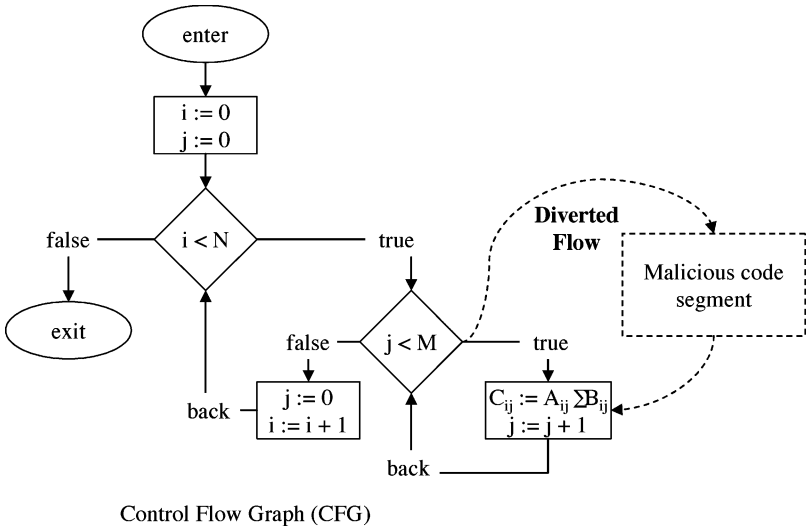
(a) Original Instruction Stream

Addr	Instruction
0x00f0	ADD r12,r12,#0xe0
0x00f4	ADD r3,r13,#0xa0
0x00f8	SUB r12,r1,r0
0x00fc	ADD r14,r13,#4
0x0100	ADD r14,r13,#4
0x0104	LDR r3,[r3,r0,LSL #2]
0x0108	LDR r2,[r2,r1,LSL #2]
0x010c	ADD r0,r0,#1
0x0110	MLA r2,r12,r3,r2

Instruction at 0x00fc repeated at request for 0x0100

(b) Instruction Stream with Replay Attack

Fig. 4. A sample instruction stream that is targeted with a replay attack.



Control Flow Graph (CFG)

Fig. 5. A sample control flow graph (CFG) with a diverted flow.

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. 5.

- **Runtime data attacks**—by examining the pattern of data writebacks to RAM, the attacker can determine the location of the runtime 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.

IV. Our Approach

At its highest level, our approach is best classified as a combination 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, our approach improves upon previous software protection attempts by accelerating their performance without sacrificing any security.

A. 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. 6). In our architecture,

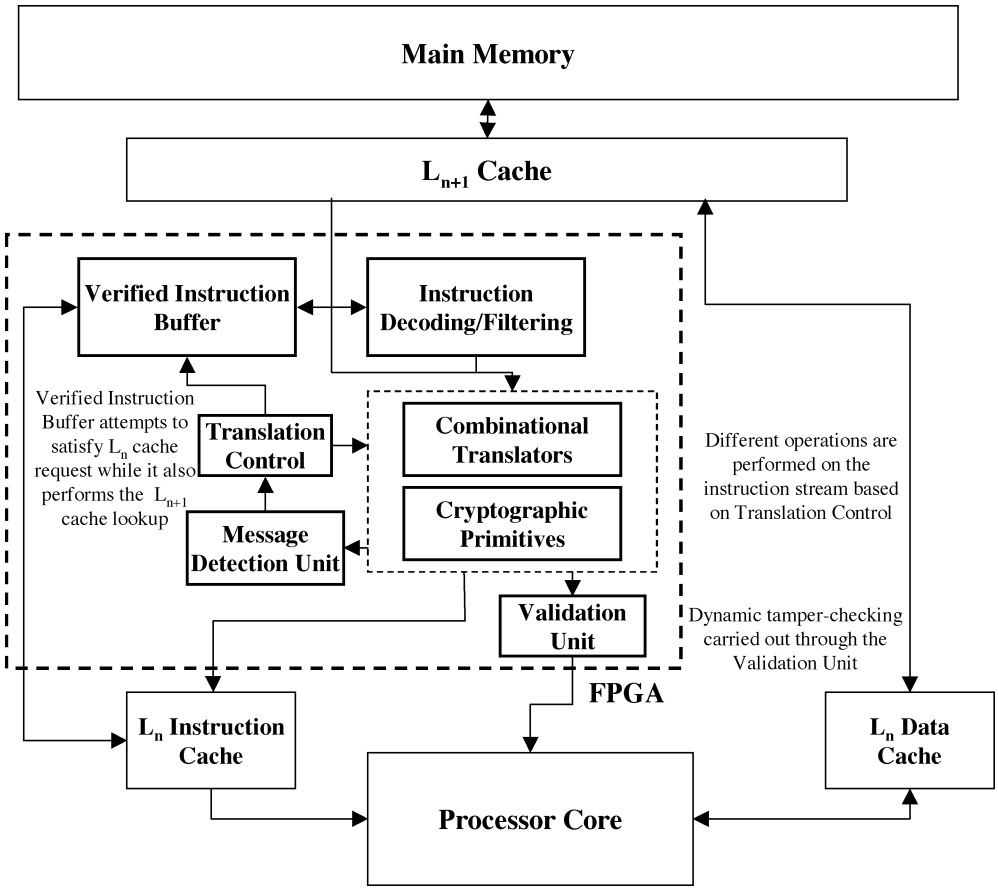


Fig. 6. FPGA-based software protection architecture.

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.

- **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 [26]. These results have shown that current-generation FPGAs are capable of extremely fast decryption. As an example in [27] an implementation of the AES cipher attained a throughput far exceeding 10 Gb/s 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.

B. 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. 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 runtime 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.

C. 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 sections, we delve into the specifics of two examples that illustrate the potential of our combined compiler/FPGA technique.

Example 1—Tamper-Resistant Register Encoding: Consider a code segment in any application and focus on the instructions as they are fetched from memory to be executed by the processor (the *instruction stream*). 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 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. As an example, the sequence of instructions $load\, x, r1 | load\, y, r2 | add\, r1, r2$ can be decoded to generate the register stream of r_1, r_2, r_2. After being extracted, the register stream is then given a binary encoding; in our example r_1 can encode $\mathbf{0}$ and r_2 can encode $\mathbf{1}$, and therefore the particular sequence of registers corresponds to the code $\mathbf{0\ 1\ 1}$.

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 and then compares 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 [24] and [28].

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.

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. Also, the register sequence can be used to encode several different items, such as authorization codes or cryptographic keys. 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. Finally, we note that when examining a block of code to be encrypted using our scheme, it will often be the case that the compiler will lack a sufficient number of register-based instruction with which to encode a suitable key. In this case, "place-holder" instructions which contain the desired sequence values, but which otherwise do not affect processor state, will need to be inserted by the compiler. In Section V-B we examine how these inserted instructions can effect the overall system performance.

Example 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. 7 shows how the compiler 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. 6. 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

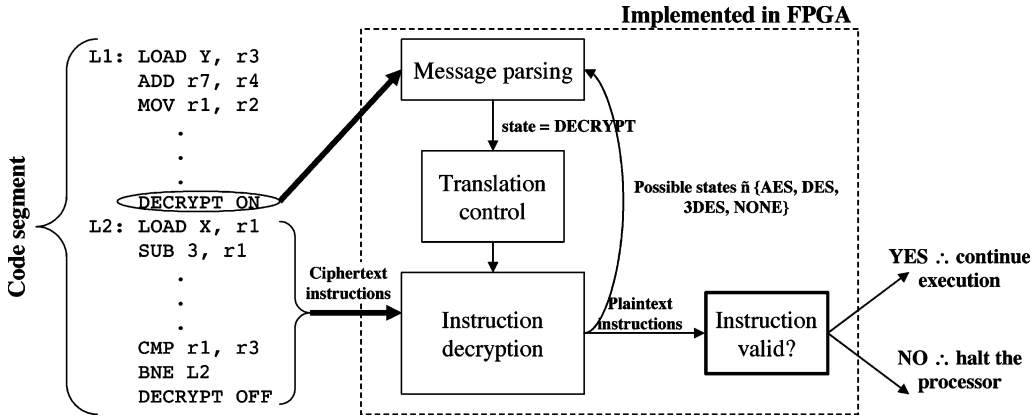


Fig. 7. Selective basic block encryption.

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.

D. 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.
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 [6], [7].
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 [29].

We have up to this point in this paper 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 section.

V. Analyzing Performance Overhead

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 cache directly from a pipelined main memory. With a fixed block size and memory bus width (in terms of number of bytes), we can estimate this penalty as the sum of an initial nonsequential memory access time for the first bytes from memory plus the delay for a constant amount of sequential accesses, which would be proportional to the product of the block size with the inverse of the memory bus width.

Now, considering a basic block of instructions of a certain length in isolation from its surrounding program, we can estimate the number of instruction cache misses as the number of bytes in the basic block divided by the number of bytes in the cache block, as each instruction in the basic block would be fetched sequentially. Consequently the total instruction cache miss delay for a single fetched basic block can be approximated as the product of the number of fetches and average fetch delay as described above.

What effect would our software protection architecture have on performance? 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. 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 any application.

A. Experimental Methodology

For the following experiments, we incorporated a behavioral model of our software protection FPGA hardware into the SimpleScalar/ARM tool set [30], 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-B lines. With no secondary level of cache, our nonsequential memory access latency is 100 cycles and our sequential (pipelined) latency is two cycles. We inserted our protective techniques 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 [31] 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 International Telegraph and Telephone Consultative Committee (CCITT) standard. We also selected from MediaBench the benchmark *pegwit*, a program that implements elliptic curve algorithms for public-key encryption. From the MiBench [32] 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 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 (for example, Xilinx Virtex-II Pro Platform FPGAs [29]). 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.

B. Initial Results

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 Section III-B. In this configuration, the compiler manipulates sequences of register-based instructions to embed codes into the executable which are verified at runtime 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 require one clock cycle, and that verifying an individual basic block by comparing a function of the register sequence with a function of the filtered instruction requires three clock cycles.

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 some instructions into the basic block which encode a portion of the desired sequence but which otherwise do not affect processor state. However, as these "place-holder" 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. Fig. 8 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 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.

Fig. 8 shows 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 runtime. 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 nonstall 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 blocks.

In Fig. 9, we 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 aggressive configuration, 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 AES imple-

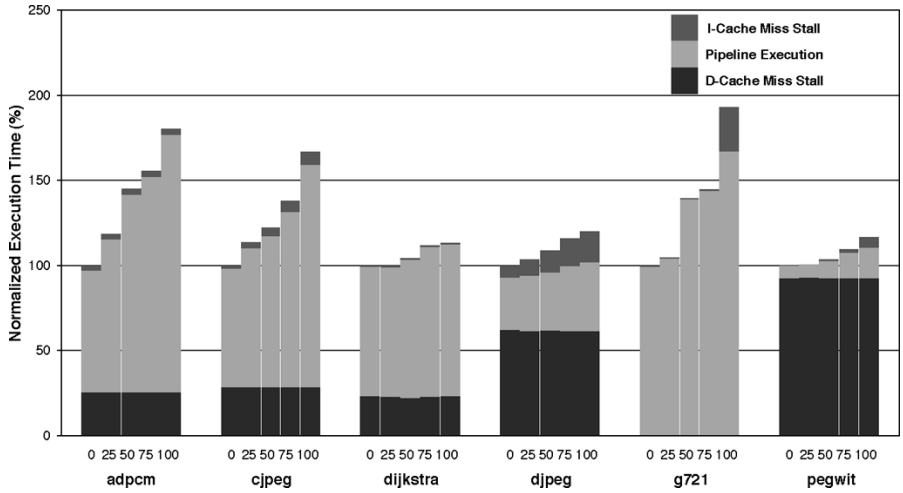


Fig. 8. Performance breakdown of the tamper-resistant register encoding scheme as a function of the basic block select rate.

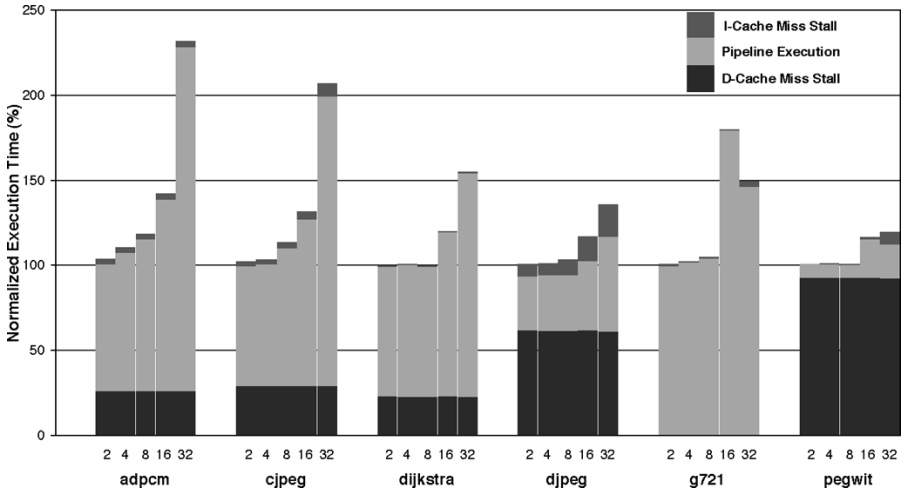


Fig. 9. Performance breakdown of the tamper-resistant register encoding scheme as a function of the register sequence length.

mentations have reached a high throughput by unrolling and pipelining the algorithmic specification [27]. The resultant pipeline is usually quite deep. Consequently 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 a 50-cycle nonsequential access and a single cycle sequential access delay. Note that other AES implementations concentrate on optimizing other characteristics besides throughput (i.e., area, latency, throughput efficiency), but an examination of the performance impact of those designs on our architecture is outside the scope of this paper.

Fig. 10 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 overall 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 Fig. 8 with Fig. 10 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 cryptographic solution that has a much more measured impact on performance. While the results in this section show that the instruction miss penalty is not 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 possibility

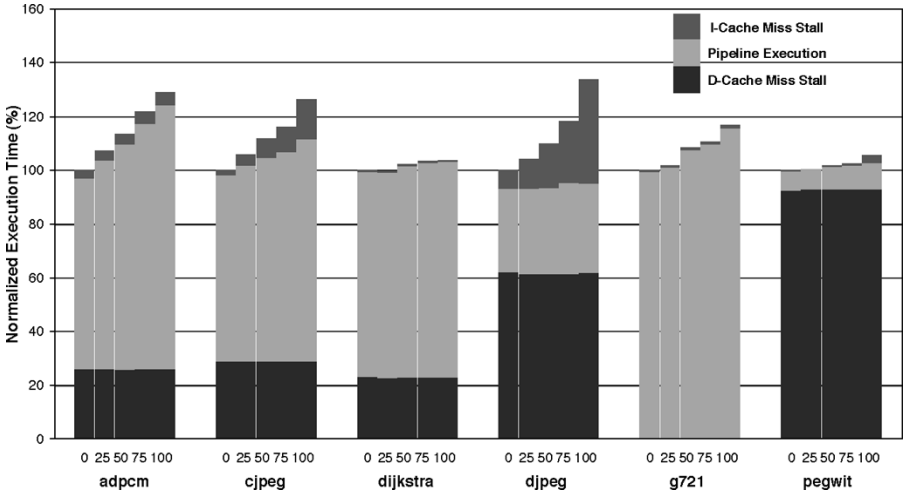


Fig. 10. Performance breakdown of the selective basic block encryption scheme as a function of the block select rate.

that programmable FPGA resources will not all be allocated for software protection purposes motivates the configuration of these resources as performance-oriented architectural optimizations.

VI. 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: 1) do not provide enough security; 2) are prohibitively slow; or 3) 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 predefined 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.

References

[1] International Plannning and Research Corporation, Eighth annual BSA global software piracy study [Online]. Available: http://www.bsa.org Jun. 2003

[2] Computer Security Institute and Federal Bureau of Investigation, CSI/FBI 2002 computer crime and security survey [Online]. Available: http://www.gocsi.com Apr. 2002

[3] C. Collberg and C. Thomborson, "Watermarking, tamper-proofing, obfuscation: Tools for software protection," *IEEE Trans. Softw. Eng.,* vol. 28, no. 8, pp. 735–746, Aug. 2002.

[4] ——, "Software watermarking: models and dynamic embeddings," in *Proc. 26th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages* 1999, pp. 311–324.

[5] R. Venkatesan, V. Vazirana, and S. Sinha, "A graph theoretic approach to software watermarking," in *Proc. 4th Int. Information Hiding Workshop* 2001, pp. 157–168.

[6] H. Chang and M. Atallah, "Protecting software code by guards," in *Proc. ACM Workshop on Security and Privacy in Digital Rights Management* Nov. 2000, pp. 160–175.

[7] B. Horne, L. Matheson, C. Sheehan, and R. Tarjan, "Dynamic self-checking techniques for improved tamper resistance," in *Proc. ACM Workshop Security and Privacy in Digital Rights Management* 2001, pp. 141–159.

[8] D. Aucsmith, "Tamper-resistant software: An implementation," in *Proc. 1st Int. Workshop Information Hiding* 1996, pp. 317–333.

[9] G. Wurster, P. van Oorschot, and A. Somayaji, "A generic attack on checksumming-based software tamper resistance," in *Proc. IEEE Symp. Security and Privacy* 2005, pp. 127–138.

[10] G. Necula, "Proof-carrying code," in *Proc. 24th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languagers* 1997, pp. 106–119.

[11] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Dept. Comput. Sci., Univ. Auckland, Tech. Rep. 148, Jul. 1997.

[12] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," in *Proc. Advances in Cryptology (CRYPTO '01)* pp. 1–18.

[13] S. White, S. Weingart, W. Arnold, and E. Palmer, "Introduction to the Citadel Architecture: Security in physically exposed environments," IBM Res. Div., T.J Waston Res. Ctr., Tech. Rep. RC 16682, May 1991.
[14] D. Tygar and B. Yee, "Dyad: A System for Using Physically Secure Coprocessors," Dept. Comput. Sci., Carnegie Mellon Univ., Tech. Rep. CMU-CS-91-140R, May 1991.
[15] S. White and L. Comerford, "ABYSS: a trusted architecture for software protection," in *Proc. IEEE Symp. Security and Privacy* 1987, pp. 38–51.
[16] B. Schneier and A. Shostack, "Breaking up is hard to do: Modeling security threats for smart cards," in *Proc. USENIX Workshop on Smartcard Technology* 1999, pp. 175–185.
[17] P. Clark and L. Hoffman, "BITS: A smartcard protected operating system," *Commun. of the ACM,* vol. 37, no. 11, pp. 66–70, Nov. 1994.
[18] MIPS, SmartMIPS Application-Specific Extension [Online]. Available: http://www.mips.com 2004
[19] VIA, Padlock Hardware Security Suite [Online]. Available: http://www.via.com 2004
[20] Intel, Intel LaGrande Technology [Online]. Available: http://www.intel.com 2004
[21] ARM, ARM TrustZone Technology [Online]. Available: http://www.arm.com 2004
[22] Trusted Computing Group [Online]. Available: http://www.trustedcomputing.org, 2003
[23] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," in *Proc. 9th Int. Conf. Architectural Support for Programming Languages and Operating Systems* 2000, pp. 168–177.
[24] D. Kirovski, M. Drinic, and M. Potkonjak, "Enabling trusted software integrity," in *Proc. 10th Int. Conf. Architectural Support for Programming Languages and Operating Systems* 2002, pp. 108–120.
[25] X. Zhuang, T. Zhang, and S. Pande, "HIDE: An infrastructure for efficiently protecting information leakage on the address bus," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems* 2004, pp. 72–84.
[26] A. Elbirt, W. Yip, B. Chetwynd, and C. Paar, "An FPGA implementation and performance evaluation of the AES block cipher candidate algorithm finalists," in *Proc. 3rd Advanced Encryption Standard (AES3) Candidate Conf.* 2000, pp. 13–27.
[27] K. U. Jarvinen, M. T. Tommiska, and J. O. Skytta, "A fully pipelined memoryless 17.8 Gbps AES-128 encryptor," in *Proc. Int. Symp. Field Programmable Gate Arrays (FPGA)* 2003, pp. 207–215.
[28] J. Zambreno, A. Choudhary, R. Simha, and B. Narahari, "Flexible software protection using HW/SW codesign techniques," in *Proc. Design, Automation, and Test in Europe* 2004, pp. 636–641.
[29] Xilinx, Virtex-II Pro Platform FPGA Data Sheet [Online]. Available: http://www.xilinx.com 2003
[30] D. Burger and T. M. Austin, "The Simplescalar Tool Set, Version 2.0," Dept. Comput. Sci., Univ. Wisconsin-Madison, Tech. Rep. CS-TR-97-1342, Jun. 1997.
[31] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," in *Proc. 30th Annu. Int. Symp. Microarchitecture* Dec. 1997, pp. 330–335.
[32] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. 4th IEEE Annu. Workshop Workload Characterization* 2001, pp. 3–14.

Joseph Zambreno (Student Member, IEEE) received the B.S. degree *summa cum laude* and the M.S. degree from Northwestern University, Evanston, IL, in 2001 and 2002, respectively. He is currently working toward the Ph.D. degree in electrical and computer engineering at Northwestern University.

His most recent industry experience was a Software Test Engineer internship at Microsoft Corporation, Redmond, WA, in 2000. His research interests include computer security and cryptography, with a focus on runtime reconfigurable architectures and compiler techniques for software protection.

Mr. Zambreno has been the recipient of several honors and awards, including membership in the Tau Beta Pi and Eta Kappa Nu honor societies, a Motorola undergraduate research award, a Walter P. Murphy graduate research fellowship, and a National Science Foundation graduate research fellowship. He is a member of the IEEE Computer Society.

Dan Honbo (Student Member, IEEE) received the B.S. degree in computer engineering from Northwestern University, Evanston, IL, in 2003. He is currently working toward the Ph.D. degree at Northwestern University.

His research interests are in cryptography, software protection, and embedded system design.

Alok Choudhary (Fellow, IEEE) received the B.E. (Hons.) degree from the Birla Institute of Technology and Science, Pilani, India in 1982, the M.S. degree from the University of Massachusetts, Amherst, in 1986, and the Ph.D. degree in electrical and computer engineering from the University of Illinois, Urbana-Champaign, in 1989.

From 1989 to 1996, he was on the faculty of the Electrical and Computer Engineering Department at Syracuse University. Since 2000, he has been a Professor of Electrical and Computer Engineering at Northwestern University, Evanston, IL, where he also holds an adjunct appointment with the Kellogg School of Management in the Marketing and Technology Innovation Departments. His research interests are in high-performance computing and communication systems, power aware systems, information processing, and in the design and evaluation of architectures and software systems.

Prof. Choudhary's career has been highlighted by numerous honors and awards, including the National Science Foundation Presidential Young Investigator Award, an IEEE Engineering Foundation award, an IBM Faculty Development award, and an Intel Research Council award.

Rahul Simha (Member, IEEE) received the Ph.D. degree in computer science from the University of Massachusetts, Amherst, in 1990.

He was an Assistant Professor in the Department of Computer Science, College of William and Mary, Williamsburg, VA, from 1990 to 1996, and an Associate Professor from 1996 to 2000. Since June 2000, he has been an Associate Professor in the Department of Computer Science, George Washington University, Washington, DC. His research interests include networks, algorithms, software systems, and embedded systems.

Bhagirath Narahari received the B.E. degree in electrical engineering from Birla Institute of Technology and Science, Pilani, India, and the M.S. and Ph.D. degrees in computer and information science from the University of Pennsylvania, Philadelphia, in 1984 and 1987, respectively.

He is a Professor in the Department of Computer Science at George Washington University, Washington, DC, where he served as department chair from 1999 to 2002. His research interests include computer architecture, embedded systems, compiler optimization, power-aware computing and communications, distributed systems, and networks.