A Compiler-Hardware Approach to Software Protection for Embedded Systems

Olga Gelbart [a], Eugen Leontie [a], Bhagirath Narahari [a,*], Rahul Simha [a]

[a] *The George Washington University, Department of Computer Science, 801 22nd Street NW, Suite 702, Washington, DC 20005, USA*

Abstract

Because of their rapid growth in recent years, embedded systems present a new front in vulnerability and an attractive target for attackers. Their pervasive use, including sensors and mobile devices, makes it easier for an adversary to gain physical access to facilitate both attacks and reverse engineering of the system. This paper describes a system – CODESSEAL – for software protection and evaluates its overhead. CODESSEAL aims to protect embedded systems from attackers with enough expertise and resources to capture the device and attempt to manipulate not only software, but also hardware. The protection mechanism involves both a compiler-based software tool that instruments executables and an on-chip FPGA-based hardware component that provides run-time integrity and control flow checking on the executable code. The use of reconfigurable hardware allows CODESSEAL to provide such security services as confidentiality, integrity and program-flow protection in a platform-independent manner without requiring a redesign of the processor. Similarly, the compiler instrumentation hides the security details from software developers. Software and data protection techniques are presented for our system and a performance analysis is provided using cycle accurate simulation. Our experimental results show that protecting instructions and data with a high level of security can be achieved with low performance penalty, in most cases less than 10%.

Key words: Embedded systems, FPGA, codesign, security, software integrity, encryption.

* Corresponding author. Tel: 202-994-7181. Fax: 202-994-4875

Email addresses: rosa@gwu.edu (Olga Gelbart), eugen@gwu.edu (Eugen Leontie), narahari@gwu.edu (Bhagirath Narahari), simha@gwu.edu (Rahul Simha).

1 Introduction

Software protection is one of the significant problems in security today. An estimated $100 - $200 billion is lost in sales due to industrial espionage and $12 billion in business losses due to software protection failures [24]. In addition, while software protection is an important issue for desktops and servers, the fact that over 90% of all processors are embedded [5] underscores the importance of protecting software on embedded systems. Embedded software systems are especially vulnerable, as they tend to be written in lower-level languages with poor support for runtime error checking. Furthermore, embedded devices are more easily captured or stolen for tampering purposes, resulting in loss of intellectual property and critical secrets. Additionally, following a physical capture, a successful attack may easily be replicated because embedded processors are so numerous. Indeed, embedded systems present an ideal scenario for resourceful attackers because they can access the processors that drives the system – an attacker that wants to probe vulnerabilities in a cell phone, or media player, merely needs to extract the hardware, and after devising an attack, can easily install modified hardware components in these units.

One approach to securing embedded software from resourceful attackers is to encrypt executables so that instructions and data are decrypted within the processor chip. We term such platforms Encrypted Execution and Data (EED) platforms. This approach considerably raises the bar for attackers with hobby-level equipment, and aims to prevent attacks on confidentiality and software integrity. However, such systems can nonetheless be attacked by exploiting the structure in encrypted instruction streams and data. As discussed in [33], by identifying block level control flow of the application code the attacker can learn more about the programmer's code, secret keys and sensitive data. The goal of this paper is to devise mechanisms to protect EED platforms, and to evaluate the overhead incurred in using these mechanisms. Since many embedded systems typically have timing requirements, the overhead incurred determines the feasibility of a protection scheme. Our approach differs from prior work in that we focus on ease-of-adoption – our approach does not require custom hardware design because we use on-chip reconfigurable logic in the form of Field Programmable Gate Arrays (FPGA), and imposes no burden on software developers because the techniques are directly incorporated into the compiler. And because the compiler instrumentation is at the backend and independent of other compiler modules, it is easy to modify legacy executables to operate in our framework.

Our approach, called CODESSEAL – **CO**mpiler **D**evelopment **S**uite for **SE**cure **A**pp**L**ications – can be summarized as follows. The hardware platform we target is a standard processor with an accompanying on-chip reconfigurable

logic core – this enables us to leverage commercially available System-On-Chip (SOC) architectures. Consider an application that is to run on this platform. First, the compiler uses the control-flow graph built during compilation to embed special labels and integrity information into each block of the executable, after which the compiler encrypts the executable with a pre-determined session key at a secure location prior to distribution. In a similar manner the reconfigurable logic is programmed with what we call the "guard" – a hardware component that contains the session key, is able to decrypt executables, and check the integrity information embedded in the executables. At runtime, when the processor fetches blocks into cache, the blocks are routed through the guard, which performs decryption and checks integrity, and which is able to detect the types of EED attacks described in this paper. Similarly, when data is written to memory, the guard encrypts the data while also embedding sufficient additional information to detect certain attacks. By relying on the secure hardware component, *i.e.,* the FPGA guard, our approach can accelerate the execution of encrypted programs in a secure environment thereby incurring acceptable performance penalties, and without requiring new processor designs. Additionally, it provides flexibility, through reprogramming of FPGAs, to carry out application specific compiler-driven protections of the application code and data.

The rest of the paper is organized as follows. Section 2 discusses previous work in hardware assisted EED platforms. Section 3 describes the threat and attack model considered in this paper. Section 4 presents the CODESSEAL architecture in detail. It describes the hardware architecture, the compiler instrumentation tasks, and the security mechanisms. Section 5 provides a security analysis of our solutions and discusses our assumptions. Section 6 experimental results and a performance analysis of our system and Section 7 concludes the paper.

2 Related Work

The general area of computer security, and in particular, software protection, has grown tremendously over the past decade and a result there is a significant amount of literature that addresses various aspects of software security. We restrict ourselves to reviewing related work in compiler-hardware approaches, and in FPGA-related work in the area of security.

The use of digital signatures at compile time, to help identify whether application code has been modified, has been the topic of several papers [2,13,3]. The work in [2] introduces the concept of guards, pieces of executable code that perform checksums to guard against tampering. In [13], 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 reports modifications. In [3] a self-checking technique is presented in which embedded code segments verify integrity of the program during runtime. These self-checking approaches [2,3,13] essentially compute checksums on code to assert code integrity. This computation is exactly the same as any other digest or MAC computation for secure communication: it relies on the high probability that a modification to the code will create a modified checksum. Such digest checking is attractive because digests can detect any kind of modification to code or data, and is relatively inexpensive in terms of computation effort. At the same 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. However, 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 [13]. More importantly, in the context of this paper, these techniques do not protect against physical attacks where the adversary has captured the device.

Hardware approaches can be categorized into co-processor solutions [28,22], including smartcard applications [15], and solutions that specify particular architectures or use FPGA's as hardware accelerators. FPGA's have been used to implement accelerated versions of several well-known cryptographic primitives such as private-key algorithms, secure hash algorithms, and public-key algorithms [7,10,17]. Some of the recent work in this area has focused on implementing high-throughput or low-area Symmetric key Block Cipher (SBC) architectures on FPGAs [31].

Among architectures specifically designed for software protection, there is past work that on memory protection [5,25], on specific attacks [20], or even the initialization of a system [1]. Our own work in this area [29,30] has focused on using compiler-directed register allocation to embed watermarks that are then checked in FPGA support hardware.

A subclass of hardware approaches are those directed at EED platforms and embedded platforms [18]. Among the first of these is the XOM architecture [16] in which instructions stored in memory are encrypted and the XOM CPU decrypts before execution. Nonetheless, attacks are possible on EED platforms and therefore a number of papers have focused on addressing such attacks. The SPEF [14] architecture presents a new processor that provides tamper-resistant environment. The AEGIS [23] architecture presents techniques for control-flow protection, privacy, and prevention of data tampering. Their techniques work at a cache block level, and require processor re-design. In [32,33],

the authors study the problem of information leakage when an attacker extracts patterns of access in an EED platform and matches those patterns against a database of well-known patterns extracted from open-source software or from unencrypted executables run inside a debugger. Their findings suggest that many algorithms can be identified by observing their memory access pattern and that this signature pattern can itself lead to both information leakage as well as additional types of attacks. They propose address randomization to foil such attacks and study the performance of specific architectural support hardware for address randomization. Finally, our own work in this area [29,30] has explored the use of reconfigurable logic, in the form of an FPGA, towards providing software security. In [29] we presented a compiler-FPGA approach to watermark detection but does not address EED attacks.

A common theme in past work on preventing EED attacks is the design of new processors. These proposed changes to the processor (and instruction set) requires a buy-in from chip manufacturers and does not leverage COTS technology. In addition, developing security schemes that are architecture dependent reduces portability. These factors serve as key motivation for our approach – we explore solutions that do not require changes to the processor and apply security measures higher up in the compilation chain to reduce the amount of architecture dependence. As we have seen in Section 2, there has been a lot of work done in the field of software protection. Our research proposes to add protection methods, which would complement already existing research.

3 Threat Model

In order to better understand the focus of the our proposed methods, it is important to see the threat model. In general, systems can be exposed to two means of attacks in terms of access to the system: *remote* attacks and *physical* attacks. In a remote attack on the application the system is accessed through some kind of a communications channel (be it a network or just an input variable) – these constitute the vast majority of attacks addressed by current work. A physical attack assumes that the attacker has direct physical access to the hardware and is sophisticated and resourceful enough to examine and manipulate instruction and data buses. We focus on physical attacks on EED platforms. However, the fact that encryption occurs in blocks enables a sophisticated attacker to mount some attacks on EED platforms, as we outline below. Even more importantly, since memory chips can be controlled externally, the attacker can supply the processor with any block of their choosing. The most effective form of attack tries to supply the processor with an unexpected block; in doing so, an attacker might then observe the outcome and use that advantageously. For example, an attacker might notice that skipping a certain block leads to skipping a license check. We consider the following

types of attacks:

- *Code Injection/Execution Disruptions*: Here an attacker tries to modify or replace a portion of an encrypted block of instructions. Of course, if we assume the key has not been broken, this attack merely places random bits into a cache block. Nonetheless, these random bits will be decrypted into possible valid instructions, whose outcome can be observed carefully by our sophisticated attacker. We can estimate the probability that randomly-injected bits result in valid opcodes. If the Instruction Set Architecture (ISA) happens to use n bits for each opcode, there are a total of 2^n possible instructions. If, among these, v is the number of valid instructions, and if the encryption block contains k instructions, then the probability that the decryption will result in at least one invalid instruction in the block is $1 - (\frac{v}{2^n})^k$. Since a good processor architecture doesn't waste opcode space with unused instructions, it is highly probable that if the attacker supplies a random block it will be decrypted and executed without detection. For example if we consider an encryption block size of 16 bytes and if 90% of the opcode space is used for valid instructions, the probability of an undetected disrupted execution is 19%. We term this type of attack *execution disruption through instruction/code injection*. The attacker will not be able to execute the exact code that they want to execute,but by observing the program's behavior, the attacker can deduce information about the program's execution. Such code injection EED attacks suggest the need for run-time code integrity checking, for example by the use of signatures embedded into the executable.

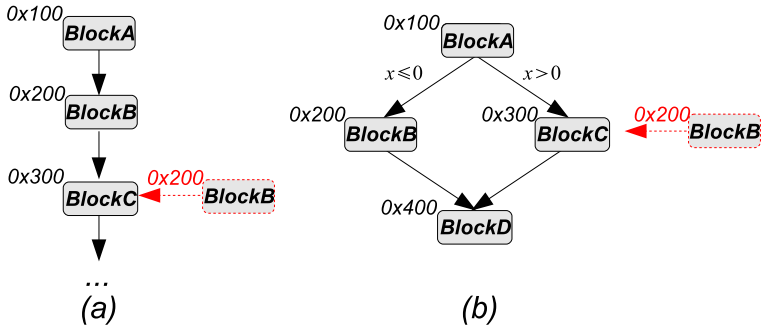

Fig. 1. Example of (a) Injection and (b) Control Flow Attacks

- *Instruction Replay*. In this type of attack, the attacker re-issues a block of encrypted instructions from memory. This can be accomplished either by freezing the bus and replacing the memory read value with an old one, overriding the address bus with a different memory location than the one the processor requested or simply overwriting the memory at the targeted address. This is illustrated in Figure 1(a). In this example, the processor requests blocks A, B, C from addresses $0x100, 0x200, 0x300$ respectively. However, when the processor requests block from address $0x300$, the attacker

injects block B (which they know will be correctly decrypted and executed). What is clear is that the incorrect block is decrypted into valid executable code. If the replayed block has an immediate observable result (such as an I/O operation) the attacker can store the block and replay it at any point of time during program execution, without the attacker having to guess the entire instruction block functionality. Also, by observing the results of a multitude of replay attacks, the attacker can catalog information about the results of individual replay attacks and use such attacks in tandem for greater disruption or to get a better understanding of the application code. The vulnerability of EED platforms to such replay attacks suggests the need to validate that the contents are indeed from the memory location requested by the processor. We note that simply storing the signature inside the code block does not prevent such attacks.

- *Control Flow attacks.* Since the attacker has access to the address and data bus, by sniffing on the address bus they can elucidate the control-flow structure of a program without any decryption. This control-flow information can lead to both information leakage as well as attacks on the application. As described in [32,33], obtaining the block level control flow graph (CFG) can lead to information leakage since control flow graphs can serve as unique fingerprints of the code and can detect code reuses. By watching the interaction and calling sequences, the attacker can learn more about the application code. Additionally, as pointed out in [32,33], knowledge of the CFG can also compromise a secret key and leak sensitive data. Since branches compare values, the attacker could force which path to take in the application code. To disrupt the execution or steer the execution in the desired direction, the attacker can transfer control out of a loop, transfer control to a distant part of the executable (thus allowing them to circumvent license checks), or force the executable to follow an existing execution path (akin to subverting an if-then-else statement). For example, consider Figure 1(b). During normal execution, block A (at address 0x100) can transfer control to either block B (address 0x200) or block C (address 0x300) depending on the value of some variable x. An attacker who has observed this control flow property, can substitute block C when B is requested and observe the outcome as a prelude to further attack. Thus, they can successfully bypass any check condition that may be present in block A. Additionally, another type of attack is when blocks A and B together form a loop. Then, upon observing this once without interference, and recording the blocks, the attacker can substitute blocks from an earlier execution to prevent the loop from being completely executed. As we have observed above, simply signing and encrypting code blocks is not sufficient to prevent control flow attacks. What is needed is a mechanism by which the correct control flow, as specified by the application, must be embedded and validated at run-time.
- *Data injection/modification.* In a manner similar to attacks on application code, discussed thus far, the application data is also susceptible to EED attacks. By examining the pattern of data write-backs to RAM, the attacker

can intelligently guess the location of the runtime stack even if that is encrypted, as commonly-used software stack structures are relatively simple. Since the attacker has physical access to the device, he/she can try to inject their own data and (even though the data will be decrypted into a random stream) observe the program's behavior. The attacker may still be able to disrupt the execution or learn something about the executable.

- *Data substitution.* The attacker can substitute a currently-requested data block for another block, which is also currently valid and observe the program's behavior. Unlike instructions which are limited to the valid opcodes in the instruction set, any data injected by the attacker will be correctly decrypted and passed to the processor. Thus, encryption in this case provides no protection other than privacy of the application's data.

As described above, these EED attacks point out that mere encryption is not sufficient to guarantee proper execution and that these types of attacks can go undetected unless we provide explicit support. We now turn to our approach in which a combination of compiler-inserted information and supporting hardware forms the framework needed to detect and prevent such attacks.

4 The CODESSEAL System

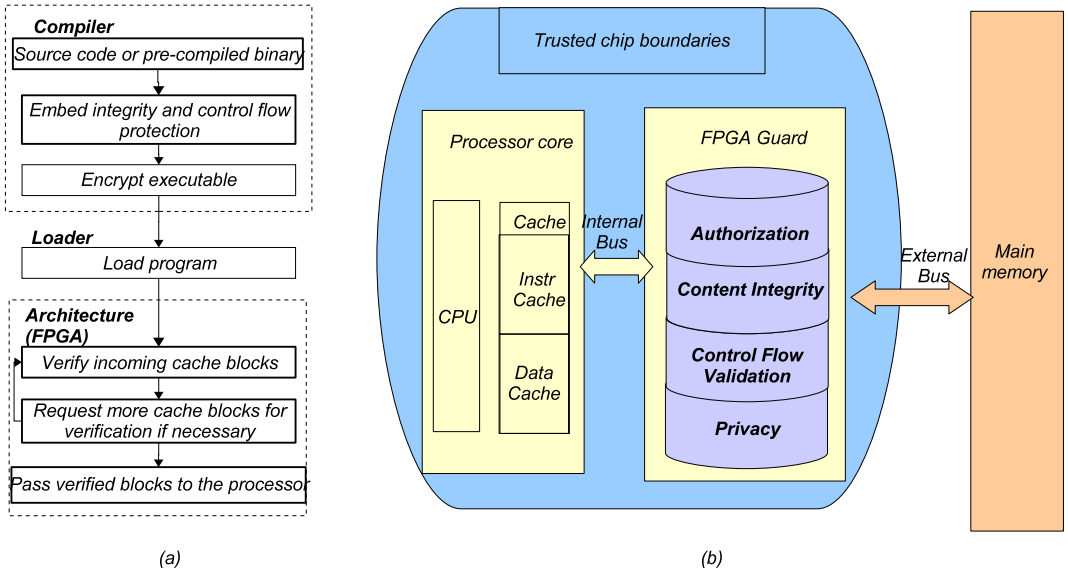

Fig. 2. (a) The CODESSEAL Infrastructure and (b) Hardware Architecture

4.1 Overview of our Approach

Our approach, designed for a standard Harvard architecture (with separate instruction and data memory), has three core components. The first is architectural: the use of supporting FPGA hardware that we refer to as the *FPGA-Guard*. The second is a back-end compiler module that instruments the executable such that each code block has a label. The third is a detection algorithm, implemented on the FPGA, that examines the labels of code blocks to verify proper execution (control flow and code integrity checking). Figure 2(a) illustrates our overall framework and the specific tasks carried out by the FPGA and compiler components. Figure 2(b) illustrates our hardware architecture.

The key elements of our approach are as follows:

- The FPGA is programmed and configured with logic to run in a co-processing paradigm.
- The FPGA contains a secure independently verifiable PKI-capable component. The configuration for the FPGA (*i.e.,* the FPGA program) is itself encrypted when downloaded using PKI. The secure loader decrypts and programs the FPGA. We will assume that the FPGA itself is verifiable through independent means.
- The compiler will generate code that is encrypted using an application key. For any code to be executed on the processor it must first be decrypted and validated by the FPGA.
- Based on program analysis, the compiler will generate integrity and control flow checking information that will be embedded into the executable.
- The FPGA encrypts the data blocks directly using the application key.

Figure 2(b) illustrates our architecture overview. The processing chip is on the left and main memory on the right, with the address and data bus connecting the two components. We assume that the chip comes with FPGA logic, as do many commercial processors today. We use this logic to implement the *guard*. To see how this works, consider how memory accesses take place without such guard logic: when a cache miss occurs, the memory management logic (in this the cache controller) issues a read to memory on the bus, after which, following the bus protocol, the memory dumps the contents on the bus. These bits are then routed into the instruction cache. Our architecture is constructed so that every read access to memory also goes through the FPGA guard. Thus, the guard logic is aware of the address requested (and the start address of an instruction block). Furthermore, in our architecture, the bus lines are routed through the guard so that the guard receives memory contents before the processor. The guard logic is therefore able to enforce our security mechanisms, such as decryption and integrity checking (which may require examining some

of the contents of the data or instructions read from memory) before it is fed into the instruction cache. Thus, all communication into and out of the chip must go through the FPGA guard thereby enabling the decryption and any other security schemes that are built into the guard. This is the key to ensuring trust: the blocks that reach the processor cache have been verified by the guard so that the processor sees (and therefore executes) only validated blocks.

In contrast to much of the past work, for example [23,16,32,33], our protection schemes operate at the granularity of a *basic block* of code; *i.e.,* our system operates on the control flow graph where each node of the graph is a basic block. By operating on basic blocks we deal with control flow properties intrinsic to the application and independent of architecture specific parameters such as size of cache line, and size of cache. Subsequently, our methods can be ported to any processor and could be applied to existing legacy code by constructing the control flow graph of the program.

4.2 CODESSEAL Security Scheme

Protection against EED attacks, as discussed in Section 3, will require providing privacy (of application code and data), integrity checking methods, and control flow validation.

We now describe our proposed schemes for integrity and control-flow validation, and the next section provides a security analysis wherein we describe how our approach prevents the various EED attacks. We first describe our solutions for each security requirement and then summarize the overall process in the next subsection.

4.2.1 Code and Data Integrity

As is typical in EED platforms, our system starts with the condition that instructions and data in encrypted form in the memory thereby providing privacy. Thus, an attacker snooping the bus cannot determine the actual application code or values of the application data. The computational overhead of asymmetric protocols suggests that we use symmetric key protocols for this purpose, and we use AES for encryption of data and instructions. We assume that a key exchange protocol is implemented in our system to allow an application key to be loaded into the FPGA guard. (We assume that a unique key is loaded into the FPGA, and its public key is available to the trusted code developers.)

Our instruction block granularity is a basic block of code – *i.e.,* a straight

line sequence of code with one entry point and one exit point (a branch or jump instruction). Since the size of a basic block can be larger or smaller than a 128-bit AES block or/and a cache block, a basic block may consist of a number of AES blocks or cache blocks. Fetching a basic block of instructions could therefore require pre-fetching a number of cache blocks (or cache lines) and decrypting a number of AES blocks. This pre-fetch logic is built into the guard. The role of the compiler in the encryption process is to generate the encrypted executables, and this is done post-compilation.

4.2.2 Code and Data Injection

As noted earlier, simply encrypting instructions and data does not prevent EED attacks. To prevent and detect code or data injection attacks we need to provide integrity checking. Embedding an integrity checking information inside each code block (in our case, a basic block) and data block (in our case, a cache block) would prevent an attacker from successfully injecting their code or data. For example, we could use any kind of hashing algorithm to generate a hash that could serve as the unique signature to embed inside each block of code or data. One option is the use of a lightweight CRC, and an alternate option is to generate a collision resistant The code block and hash is then encrypted by the compiler. The guard will then fetch the block, decrypt the block, compute the SHA-1 hash of the code and compare with the signature stored in the block. This scheme will successfully detect and prevent an attacker from injecting their code or data.

4.2.3 Control-Flow Integrity

Consider the example in Figure 3 which shows a piece of application code with two basic blocks bb_1 and bb_2 at addresses $0x00$ and $0x10$. If the branch, at address $0x0C$, is successful then the processor requests a basic block at address $0x30$. The attacker can inject basic block bb_2 at address $0x10$. Simply storing signatures does not prevent this attack on the control flow of the program since the guard will validate the signature stored in bb_2 since no instructions inside bb_2 are changed by the attacker. What is required is a mechanism for checking that the memory contents fetched are indeed from the memory located at the address requested by the processor. In other words, we need to embed the memory address into the code block signature.

We use the relative address labels of the basic block as the control-flow integrity checking information. The labels, which are relative addresses in the program, of the basic blocks are generated at the last pass in the compilation stage, and these labels are embedded into the basic block by the compiler before encryption. For example, in Figure 3(a) the label for basic block bb_1 is

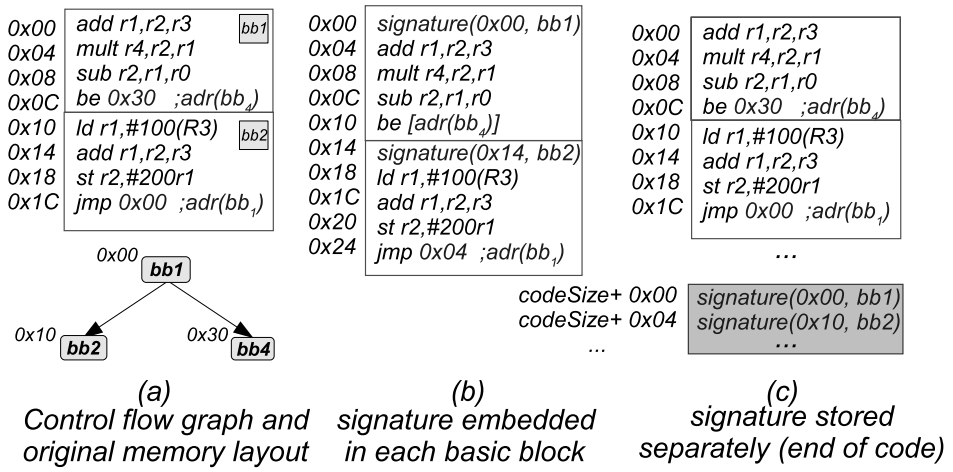

Fig. 3. Sample Code and Embedding Control Flow Information

$0x00$ and for bb_2 the label is $0x10$. Observe that the absolute address requested by the processor depends on the label of the block and the starting address of the program. However, given the starting address a_{start} of the program and the address requested by the processor a_i, we can compute the label of the requested block bb_i as $l_i = a_i - a_{start}$. The FPGA guard architecture needed to support this integrity check must read and store the address a_i requested by the processor, store the starting address of the program a_{start}, and check if the label x stored inside the block is equal to $a_i - a_{start}$. This also implies that a key assumption in our approach is that we have a secure loader, which loads the starting address into the FPGA guard – an assumption that is reasonable for most single application embedded systems such a media player or set-top box.

Based on the discussions thus far, we require some form of "signature" (to prevent code modification) and the basic block label to be embedded into the instruction blocks, *i.e.,* the compiler must embed this information into the code block so that it contains integrity and control-flow information. This is then followed by encryption thus providing a signature for the code block. Inserting this integrity information, containing a hash of the code and the basic block labels, will require re-computing the labels. Figure 3(b) shows the effect of inserting the labels (*i.e.,* a signature of length 32 bits which is one instruction) into the basic blocks; observe that basic block bb_2 now has a label of $0x14$.

What happens once this code is decrypted and validated by the guard? Since this integrity checking information, *i.e.,* the signature, is not part of the executable, the guard must replace this information with NOPs to the processor. The length of this information determines the number of NOPs inserted by the guard. For example, a 32-bit CRC is replaced by a single NOP in a 32-bit processor whereas a 160-bit SHA-1 hash results in 5 NOPs inserted into the executable sent to the processor. The number of NOPs add to the execution

time and thus constitute a performance penalty. In addition, the rearrangement of code could lead to changes in the cache performance such as the cache miss rates. An alternative to storing the signatures inside the code block is to store them in a separate portion of memory. Figure 3(c) illustrates this option for the code shown in Figure 3(a). Storing the signatures in a separate portion of memory (for example after the end of the program) implies that the guard will no longer need to insert NOPs into the executable thus eliminating the overhead of executing these NOPs. However, the guard now needs to generate an additional memory access to fetch the signature of the requested block. This also involves implementing the mapping to compute this address based on the requested address. For example, one possible mapping is to compute the address of the signature as the basic block number added to the end address of the program. Alternately, the loader could reserve a special portion of memory and specify the starting address of this part of memory to the guard. Since the decryption algorithm will take a large number of cycles the additional memory access latency can be largely hidden by overlapping the decryption with the memory fetch. Regardless of which mapping scheme we implement, we observe that the reduction in performance overhead comes with an increase in the architecture complexity (and thus in increased number of logic gates required) in the FPGA. The amount of logic available on current FPGAs, such as the Virtex II or Virtex IV systems from Xilinx Corporation, justifies this assumption [26].

We now address the algorithms used to create the signatures for each code block. We considered two methods. In the first, we insert the label of the basic block and the SHA-1 hash of the resulting instructions into the basic block. The combined size of this signature is now 192 bits (or 6 NOPS in a 32-bit processor). The performance overhead due to computation of SHA-1 [11] and due to the NOPs could result in unacceptable time penalties. When timing requirements are stringent we need to consider faster schemes that compromise security for performance. We describe such a scheme next.

To improve the performance of control-flow integrity checking,for every basic block the instructions in the block and its label are used as input to a CRC.We chose a 32-bit CRC since it is the length of one instruction; in general, we can choose the length to be equal to the processor instruction width or any other number. The CRC is then added into the basic block in a manner similar to how the SHA-1 hash would be added. We program the FPGA guard to compute the CRC, and this can be done in a single FPGA cycle. During run-time, we can compute the basic block's label as before when given the requested address and the starting address of the program. After decryption, the guard then calculates the CRC over the label and the instructions. If the computed CRC matches that stored inside the fetched block then the integrity and control-flow is validated and the block is sent to the processor. If the CRC was stored inside the basic block, then the guard will need to replace these

bits with the NOP bits.

4.2.4 Instruction and Data Replay.

Embedding just a signature into the code, or data, will not prevent an attacker from replaying an instruction. For example, consider the sample code in Figure 3(a). After basic block bb_1 has been fetched and executed, the attacker knows that instructions at address $0x00$ are valid. Thus, injecting the block at address $0x00$ when processor requests a different address will lead to a successful attack since the signature (of the injected block bb_1) is verified by the guard. The protection scheme proposed for control-flow integrity also prevents instruction replay attacks, and thus no separate mechanism is needed. Additionally, the same scheme can be used to prevent data injection attacks. In this case, the signature and cipher-text for each data block is computed by the guard and the signature is stored separately outside the data block. On memory writebacks, the guard also generates the signature and encrypts the data before writing back to memory. As with instructions, we examine the use of SHA-1 and CRC for creating the signatures for the data. However, we considered only the case where the signatures are stored external to the data block.

4.3 Architecture and Compiler Design

Our hardware-software codesign approach requires specific functions in the compiler and the hardware. The hardware functions, *i.e.,* the implementation of the guard, are built entirely on the FPGA and thus require no changes to the processor. In addition, we require the loader (assumed to be trusted) to provide the starting address of the program to the FPGA guard.

The overall CODESSEAL workflow process, with the specific functions to be implemented in the compiler, loader and FPGA components is shown in Figure 4.

The changes to the compiler, to implement the functions needed, are done entirely at the back-end of the compiler and as part of the post-compilation process. The first two steps, C1 and C2 in Figure 4, consists of constructing the control flow graph and identifying and labeling the basic blocks – in fact, typical compilers provide this information. Step C3 is an implementation of the SHA-1 or CRC hash algorithms, and step C4 computes new labels based on the new addresses. Step C5 is an implementation of the AES algorithm and uses the application key.

The loader starts by establishing a session key and downloads the application key. In step L2 it sets the program start address and sends to the guard.

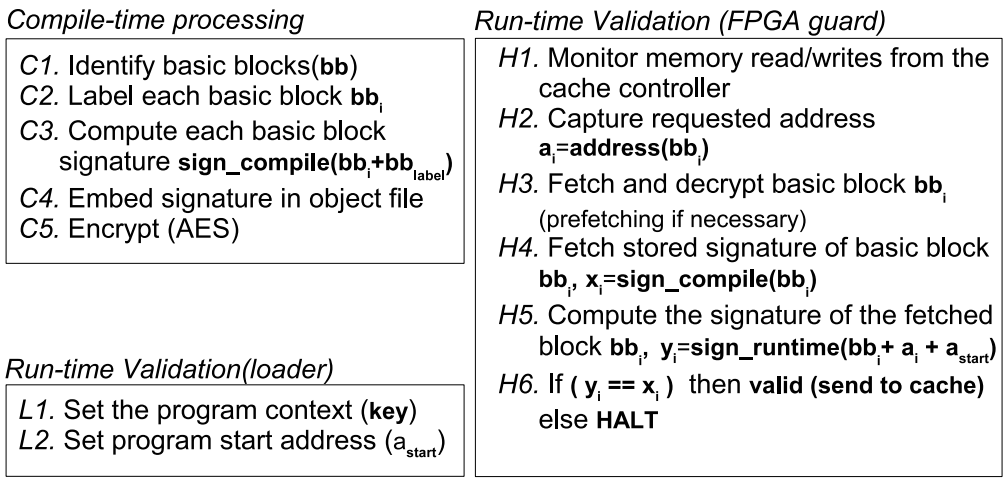

Fig. 4. CODESSEAL Process: Compiler and FPGA Roles

Much of the design effort lies in the FPGA architecture, although we leverage the opencore implementations that are available for most of the functions. The guard is designed to sit between the processor and main memory, and can thus read any memory request generated by the processor (*i.e.,* more specifically the cache controller). In the first step H1 the guard simply reads the address generated and stores it in step H2 into a local register – this can be accomplished by routing the address lines into the FPGA. Step H3 requires implementing the AES encryption and decryption algorithms. In addition, we need to implement the pre-fetch logic to fetch until the end of the basic block. This logic only requires that the guard examine the decrypted block to check if a branch instruction is present. If there is no branch then the basic block continues to the next memory location and it fetches the next cache block from memory and repeats the checking. Once the entire basic block is fetched, it moves to step H4. If the signature is stored in the basic block then there is no fetch required. However, for the case when these are stored separately it generates a request to fetch the signature. In step H5 it computes the signature based on the basic block it has fetched, and this requires implementation of the SHA-1 or CRC schemes. Step H6, the validation logic, is a simple comparison operation which can be implemented as a comparator. For the case of data, step H3 is simplified since our schemes work at a data cache block granularity. However, if the processor requested a write-back then it only executes steps H1, H2, H4, and H3 – note that it first computes signature and then encrypts the data before writing to memory. The detailed steps taken by the FPGA guard are summarized below and detailed FPGA architecture components required to implement our scheme is shown in Figure 5.

FPGA Guard Details:

- The starting address a_{start} is stored in a local register.
- Each time there is a miss in cache memory, the cache controller makes a

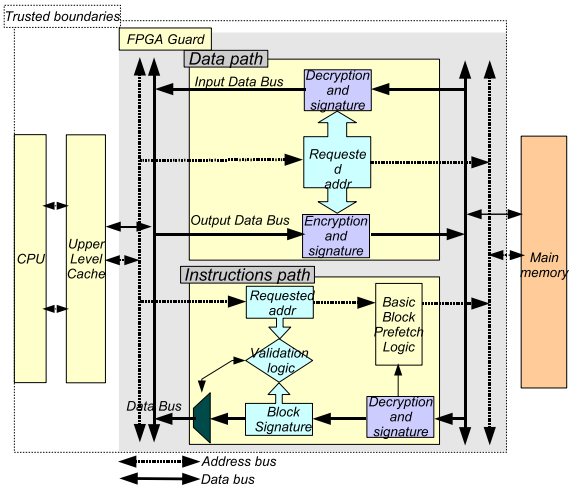

Fig. 5. FPGA Guard Architecture

request to main memory to fetch the block.

- The guard intercepts the address a_i – let's call this the *snooped address* – and stores it in a register (inside the guard).
- The guard generates all the remaining requests to memory for the rest of the block. In the meantime, the processor waits.
- When the main memory supplies the contents and when the guard has read the entire block, the guard decrypts the block using the (private) key.
- Next the guard extracts the signature and then the basic block label l_i, and checks if $a_i = l_i + a_{start}$.
- If the validation succeeds, the guard continues by checking integrity. It computes the signature (SHA1) of the fetched block and compares with the signature stored.
- If the validation or integrity check fails, the guard either stops the execution or loads a piece of code located at a predetermined static location in memory to handle the exception.
- If the validation succeeds, the guard feeds the decrypted cache block into the processor. If the signature was stored inside the block then it inserts NOP instructions.

5 Security Analysis

We first explain how our system detects the different types of attacks described earlier and then discuss the validity of the assumptions made in our approach.

First, note that code injections and execution disruptions are detected using the signature. Second, replay and control-flow attacks are detected because the guard always returns the correct block to the processor; in other words, when the cache controller requests an block (by providing the start address), the

validation mechanism ensures that no other block is delivered to the processor cache.

To see how this works, consider first a control-flow attack such as the one shown in Figure 1(b). Suppose block A transfers control to either block B or block C, depending on runtime conditions, and that block A requests (address for) block B. which the attacker remembers and stores. Later, when the attacker notices that block A requests block C, the attacker can substitute block B. However, the label for block B is different from the label of block C. The guard, by reading the address of block C and decrypting and performing the CRC, can compute the label of the requested block C. Since the labels are different, the guard guard detects the substitution and halts the processor. Similarly, in a replay attack, the attacker can substitute block A itself, which again will be caught by the guard because its label conflicts with the actual request.

Are buffer-overflow attacks detected in our system? Because buffer-overflows are considered a language vulnerability in an EED platform, neither the standard encryption nor our integrity and control flow mechanism checks against array boundaries. However typical buffer overflow attacks are detected and prevented because the code injected would have to be properly encrypted and verified. Furthermore, even if the injected code is a replay of a known encrypted block, it will not have the correct label and hence will be caught through our integrity checking mechanism.

Finally, what are the performance and security tradeoffs in choosing CRC over the SHA-1 for the hashing schemes? Let us consider the two schemes in terms of security strength; discussion of their relative performance is discussed in the next section. Assuming an EED platform with encrypted code and data, a birthday attack [21] on the hash is infeasible, since we are assuming that the encryption is strong enough. In an ideal block cipher, the probability of specific plain-text bits changing/flipping when any bit of the cipher-text is flipped is 1/2. Hence probability of flipping k bits if $(1/2^k)$ even when several bits of cipher-text are flipped. Therefore, the probability of injecting instructions, which would result in the same hash [27] is $\frac{1}{2^{160}}$. If we use a 32-bit CRC, the probability that a block modification will result in a successful CRC verification is $\frac{1}{2^{32}}$. By using a 32-bit CRC we are reducing the security strength of our scheme, but as we observe in the next section the performance penalty is greatly reduced. In general using a $k-bit$ CRC will give us a security strength of $\frac{1}{2^k}$. This then becomes a problem of a tradeoff between security and performance. We can achieve the same security strength as SHA-1 by increasing the size of the CRC to 160 bits, but this would still be potentially more efficient, since the CRC calculations are much faster than the hash.

5.1 Observations and Assumptions

Our approach makes several assumptions about the system components: compiler, operating system, and hardware. We now clarify the assumptions, and discuss further observations, for each of these three components.

- In terms of the compiler, first, we note that the labels can be inserted into executables without knowledge of the base address because only the offset is needed. Second, although we call this a post-compilation modification, it could be used directly with executables and can thus be applied to legacy code with some additional effort to extract and modify branch targets. Third, we have not explicitly provided the details of computing the integrity checksum, nor the encryption itself, because we use standard algorithms which have been addressed elsewhere. Fourth, we assume that the compiler itself is trusted since it is embedding all the protection mechanisms and performing the encryption function. This is a reasonably assumption since application code in an embedded system is developed at a trusted site before loaded to the hardware.
- In terms of the assumptions and observations about the architecture, they deal with the FPGA guard and not with the processor itself. First, we observe that the guard's actions are completely independent of the processor and require no modification of the processor's internals whatsoever. Furthermore, the manner by which the guard interacts with the processor is compatible with various cache controller algorithms such as critical-word-first or sequential-requests. Similarly, the use of the guard requires no change to main memory since the guard is programmed to use the standard bus protocol. However, what *does* change is performance: because the label is replaced by a NOP, both the size of the program and the execution time increases. By storing the label (or signature) in a separate portion of memory, we have an additional memory access to fetch this information but the code block sent to the processor does not include a NOP. The latency of this additional memory access is hidden by the latency of decryption by overlapping AES decryption with memory fetch. The cost of storing this information in a separate portion of memory is passed to the complexity of the FPGA design logic. Specifically, the guard must store the memory mapping function and generate the address where this information (signature/labels) for each address requested by the processor. Note that because the guard is implemented in FPGA logic, a variety of architectural optimizations can be explored in the future.
- Our scheme requires that the guard needs to know the base address for a program. Thus, we assume that this part of the operating system is trusted. In a simple embedded system, this assumption is quite reasonable since the load addresses are usually known ahead of time. However, a desktop system with a sophisticated operating system presents two problems: the first is

that the load address is not known prior to deployment, and the second is that the base address will need to be switched when a process is switched. Clearly, a kernel module that supplies the base addresses to the FPGA (using encrypted communication) is one way to handle this case. However, that requires a high degree of trust in the operating system.

6 Experimental Results and Performance Analysis

For the overall simulation of our system, we used the SimpleScalar simulation suite [4] for an ARM processor architecture. The performance of our architecture was observed for a memory hierarchy that contains one level separate instruction and data caches. The instruction cache has 32Kb of available 32-way associative memory. Data cache is 32Kb, 64-way associative. The simulator used is sim-outorder. The architectural features of SimpleScalar were configured to model an ARM architecture with an on-chip FPGA. In our simulation parameters, the processor speed is 400Mhz and the FPGA operates at 200Mhz. Thus every FPGA computation cycles that does not overlap processor execution creates 2 processor penalty cycles. The external bus and main memory is assumed to run at 100Mhz.

The benchmarks we have chosen from two different suites: MiBench [8], which are specifically designed for various kinds of embedded systems, and DIS [9], which are designed to be very data-intensive.

The overhead incurred by the encryption and validation mechanisms come into play only when a cache miss occurs and the requested cache block is fetched from memory and validated by the FPGA. Therefore when a cache miss occurs, the additional delays (which now consitute our performance penalties) are added to the access time to the lower level memory, *i.e.,* the cache miss penalty. However, since executable code may be rearranged and require more overall memory there could be a resulting effect on the cache miss rate of the application. Thus, the overall performance penalty is affected by three factors: increased cache misses and increased cache miss penalty, extra instruction executions due to NOPs inserted into the code, and decryption. The increase in the program size comes from reserving the extra space for the validation and this causes more cache misses, since less of the original instructions fit in the same cache memory space. The encryption and validation adds a fixed penalty for each memory fetch. The operations performed by the Guard can be modeled as an increased latency in the instruction fetch.

Figure 7 depicts how the penalty cycles were estimated for instruction fetches assuming a 32-byte cache block. We note that multiple cache blocks may need to be accessed by the FPGA pre-fetch logic in case of a basic block that spans

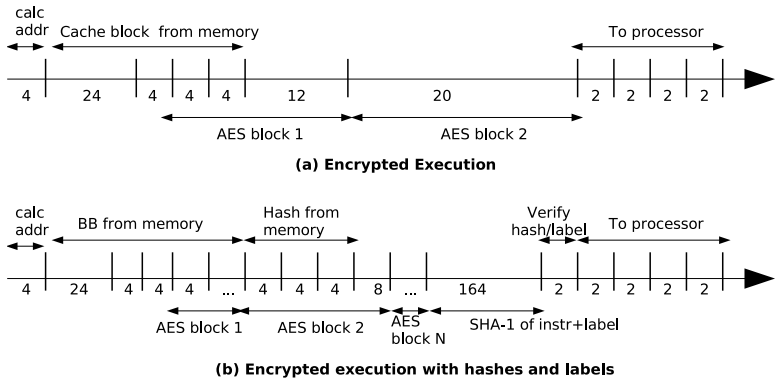

Fig. 6. FPGA Guard Details

across multiple cache blocks. Recent FPGA implementations of AES manage to achieve high throughput by pipelining the execution path and unrolling techniques [19]. The AES and SHA-1 implementations chosen by our model is one that minimizes the operational latency since high throughput is not the target in this architecture. The timing requirements for implementing these algorithms on a Xilinx FPGA were obtained from [11,12] The 10 FPGA cycles for AES [12] encryption/decryption translates into 20 processor penalty cycles that are added to the cache miss penalty. The FPGA implementation of SHA-1 takes 82 FPGA cycles which translates to 164 processor cycles. In the case where CRC is used in place of SHA-1, the guard takes 2 cycles. The other Guard processing time requirement is 1 cycle for address validation. The space utilization by the fpga guard, to implement the various algorithms, is low compared to the amount of logic available on the chips that we considered. For example, the aes-decryption takes 284 slices out of 10K available slices on a Xilinx Virtex2Pro [26].

The cache miss penalty can be computed according to the equation :

$$MissPenalty = \lceil \frac{proc_freq}{fpga_freq} \rceil * (AESDecryptionCycles + Signature - Computation + ValidationCycles) + MemAccess$$

We evaluated the effectiveness of our approach in terms of the performance overhead incurred by our techniques when compared to the execution time to the baseline case where no security mechanism is used. Figure 7 summarizes the performance penalties, for each of our security schemes, in terms of percentage increase in execution time of the application over the baseline case for each of the benchmarks. For example, the third column (labelled Encryption Labels with internal storage) for the benchmark *fft* shows that this security scheme resulted in a 13.35% increase in execution time. While we used unencrypted execution as our baseline, we note that in an encrypted system the baseline case should be when instructions and data are encrypted, *i.e.,* the baseline case should be the second column in the table (labelled "encryption"). Using the encrypted execution scheme as the baseline would result in

a smaller increase in performance penalties for our system. The performance penalties for instruction protection schemes and data protection schemes are shown separately in the Figure, under the columns for Instruction and Data categories. For protection of instructions, we analyzed the performance of integrity protection schemes using (1) CRC and (2) SHA-1 – the columns labeled "Encryption, Labels" refer to using a CRC with basic block labels and code followed by encryption, and the "Encryption,Hashing,Labels" refers to using SHA-1 along with the block labels followed by encryption. In addition, we tested each of the two instruction protection schemes for the cases where we stored the signature inside the block (shown as internal storage) and external to the block (referred to as external storage).

	Instructions					Data		
Benchmark	Encryption	Encryption, labels (internal storage)	Encryption, labels (external storage)	Encryption, hashing, labels (internal storage)	Encryption, hashing, labels (external storage)	Encryption	Encryption, labels	Encryption, hashing, labels
bitcount	0.03	9.48	0.03	47.41	0.19	0.02	0.02	0.07
crc	0.02	17.03	0.02	99.16	0.15	0.02	0.02	0.06
Dijkstra	0.02	11.57	0.03	58.02	0.54	5.27	5.50	15.62
fft	0.08	13.35	0.09	39.16	1.25	3.80	3.89	10.71
fft.inverse	0.07	14.09	0.08	47.54	3.52	12.20	12.47	33.26
patricia	0.04	44.05	0.05	1673.59	0.47	2.74	2.81	8.74
sha	0.07	5.25	0.08	22.11	0.49	0.10	0.11	0.31
stringsearch	4.95	20.54	5.69	110.48	38.30	9.75	10.23	29.63
susan.smoothing	0.04	4.95	0.05	24.60	0.31	0.16	0.17	0.49
susan.edges	0.82	5.69	0.94	18.40	5.74	6.84	7.06	19.28
susan.corners	1.50	7.80	1.73	26.24	11.55	13.00	13.41	36.17
field	0.01	2.28	0.01	11.02	0.08	0.03	0.03	0.08
pointer	0.02	6.74	0.02	33.72	0.13	0.02	0.02	0.06
tc	4.59	12.65	5.28	70.50	35.24	4.73	4.96	14.27
update	8.04	23.12	9.25	114.29	62.22	9.71	10.18	29.39
Average	**1.35**	**13.24**	**1.56**	**159.75**	**10.68**	**4.56**	**4.73**	**13.21**

Fig. 7. Performance of Instruction and Data Protection Schemes as Percentage Increase in Execution time

Examining the CRC method (*i.e.,* the column labeled "Encryption, Labels (internal storage)"), we observe performance penalties that range from 2% (field benchmark) to 44% (for Patricia). This penalty is a function of both the cache miss penalty and the extra execution cycles due to the NOPs inserted. Now consider the third column, which shows the performance of the CRC method when the signatures are stored outside the code block. As expected, due to the absence of the NOPs, the performance penalties are significantly lower than the first case and on average gives an order of magnitude improvement. However, we note that the penalties in some cases are also a result of the cache miss rate being increased. Our experiments demonstrate that internal storage of integrity information yields a higher performance penalty due to two reasons. First, as we observed in the case of *patricia*, insertion of extra NOPs resulted in a much higher cache miss rate and thus a very high performance penalty (since each cache miss incurs a miss penalty). Second, as we observed in the case of *crc*, even though the case miss rate did not significantly change,

the performance penalty still increased due to frequent execution of the extra instructions (which appear as NOPs). Overall, the average performance penalties (across all benchmarks) are 13.2% when CRC labels are stored inside the code blocks and there is an order of magnitude improvement to 1.5% when these are stored externally. In fact, the performance penalty with external storage is not significantly different from the baseline encrypted execution. Moving to the data protection columns, we observe that the average performance is comparable to just using encryption. Recall that for data protection, we store the signatures externally.

Comparing the CRC and SHA-1 signature schemes, we observe that the results are consistent with the overheads of these two schemes. As shown in column 4 of Figure 7, when the hashes are stored inside the code blocks, we incur very high performance penalties in a number of benchmarks. Once again we observed that the Patricia benchmark results in very high cache miss rates and therefore high performance penalties. As with the CRC scheme, external storage (column 5) improves performance by an order of magnitude. Overall, the average performance penalties (across all benchmarks) when the hashes are stored externally is comparable to the CRC scheme with internal storage. However, when comparing each scheme with the same storage scheme we observe that using SHA-1 results in an order of magnitude larger penalty. As can be seen from column 8 of Figure 7, similar trends hold for data protection.

In summary, protecting both instructions and data results in a performance penalty between <1% and 9% when we store CRC signatures external to the code blocks. In addition, their performance is not noticeably different from using just encryption and we conclude that our protection mechanisms are capable of protecting EED platforms with negligible overhead. We found that the cache miss rates in most of our benchmarks are very low, and therefore the miss penalty (due to decryption and signature verification) is not incurred frequently. Evaluating benchmarks, such as the SPEC benchmarks, with a higher cache miss rates will provide a better picture of how our CODESSEAL approach performs on more general applications (as opposed to embedded applications). We also observe that the performance penalties would be higher if the gap between processor speed and FPGA speed increases.

7 Conclusions and Future Work

This paper presented a joint hardware and software solution to protection of software from the class of attacks we refer to as *EED attacks*. In such attacks, even if an application's instructions and data are encrypted, an entire series of attacks on integrity and flow of control are still possible when the system is physically captured. We presented a system – CODESSEAL – which relies

on the software (compiler) side to add protection to the program and on the hardware (FPGA) side to dynamically verify the software and its data at runtime with low performance penalties. We examined different security schemes, which represent tradeoffs in security and performance, and different architecture implementations for these. Our simulation studies show that some of our security schemes incurred very low performance penalties when compared to the baseline case of encrypted execution. Based on this observation we conclude that strong integrity and control-flow protections can be provided in our system without adversely affecting any timing constraints in the embedded system. Our approach is transparent to software developers. In contrast to past approaches, our approach augments existing processor with the on-chip FPGA logic and thereby does not require making changes to the processor design, As a result we believe our approach can be applied to any embedded system. We have currently tested our scheme on a suite of benchmarks, designed specifically for embedded systems. Our future plans include testing it on larger industry-accepted benchmarks, such as the SPEC benchmarks, and exploring architectural optimizations.

Acknowledgments

The authors wish to thank Prof. Joseph Zambreno for his suggestions. This research is partially supported by NSF grant ITR-0325207 and AFOSR grant FA955006-1-0152.

References

[1] W. Arbaugh. A Secure and Reliable Bootstrap Architecture. *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 65–71, May 1997.

[2] H.Chang and M.J.Atallah. Protecting software code by guards. ACM Workshop on Security and Privacy in Digital Rights Management, Philadelphia, 2001.

[3] D. Aucsmith. Tamper resistant software: An implementation. in Anderson,R., Ed., Information Hiding, First International Workshop, Cambridge, UK, 1996, Springer-Verlag Lecture Notes in Computer Science, Vol. 1174, pp. 317333.

[4] T. Austin, E. Larson, and D. Ernst. *Simplescalar: an infrastructure for computer system modeling*, Computer (Feb 2002).

[5] M. Corliss, E. Lewis, and A. Roth. Using DISE to Protect Return Addresses from Attack. *Proceedings of the Workshop on Architectural Support for Security and Anti-Virus*, October 2004.

[6] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. *Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks*, USENIX Security Symposium (1998).

[7] A. Elbirt, W. Yip, B. Chetwynd, and C. Paar. An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists. *The Proceedings of the 3rd Advanced Encryption Standard (AES3) Candidate Conference*, pp. 13–27, 2000.

[8] 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*, IEEE 4th Annual Workshop on Workload Characterization (2001).

[9] J. W. Manke , J. Wu. Data-Intensive System Benchmark Suite Analysis and Specification. Atlantic Aerospace Electronics Corp., June 1999

[10] A. Hodjat and I. Verbauwhede. A 21.54 Gbits/s Fully Pipelined AES Processor on FPGA. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 308–309, 2004.

[11] Helion Technology: Datasheet: High Performance AES(Rijndael) cores for Xilinx FPGAs, www.heliontech.com, 2005.

[12] Helion Technology: Datasheet: Fast SHA-1 Hash core for Xilinx FPGAs, www.heliontech.com, 2005.

[13] B. Horne, L. R. Matheson, C. Sheehan, and R. E. Tarjan. Dynamic self-checking techniques for improved tamper resistance. ACM Workshop on Security and Privacy in Digital Rights Management, November 2001.

[14] D. Kirovski, M. Drinic, M. Potkonjak: Enabling trusted software security. *Proc. ACM Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS),* 2002.

[15] O. Kommerling and M. Kuhn. Design Principles for Tamper-Resistant Smartcard Processors. *Proceedings of the USENIX Workshop on Smartcard Technology,*, May 1999.

[16] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, *Architectural support for copy and tamper resistant software*, ASPLOS (2000).

[17] N. Mentens, S. Ors, and B. Preneel. An FPGA Implementation of an Elliptic Curve Processor over GF(2m). *Proceedings of the ACM Great Lakes Symposium on VLSI (GLVLSI)*, pp. 454–457, 2004.

[18] M. Milenkovic, A. Milenkovic, and E. Jovanov, *Hardware support for code integrity in embedded processors*, CASES (2005).

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

[20] H. Ozdoganoglu, C.E. Brodley, T.N. Vikaykumar, and B.A. Kuperman. *Smashguard: A hardware solution to prevent attacks on the function return address*, CACM (2005).

[21] RSA Security, "Crypto FAQ", http://www.rsasecurity.com/rsalabs/.

[22] S. Smith and S. Weingart, Building a High-Performache Programmable Secure Coprocessor, *Computer Networks*, Vol. 31, pp. 831–860, 1999.

[23] G.E. Suh, D. Clarke, B. Gassend, M van Dijk, S. Devdas: AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing. *Prof. ICS 2003.*

[24] D. Wagner, J.S. Foster, E.A. Brewer, A. Aiken, *A first step towards automated detection of buffer overrun vulnerabilities,* Network and Distributed System Buffer-Overflow Symposium, 2000.

[25] E. Witchel, J. Cates, and K. Asanovic. Mondrian Memory Protection. *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[26] Xilinx Corporation. www.xilinx.com.

[27] X. Wang, Y. Yin, H. Yu, "Finding Collisions in the Full SHA-1", *Proceedings of the* 25^{th} *Annual International Cryptology Conference*, Santa Barbara, CA, August 2005

[28] B. Yee and J. Tygar. Secure Coprocessors in Electronic Commerce Applications, *Proceedings of the USENIX Workshop on Electronic Commerce*, pp. 155–170, July 1995.

[29] J. Zambreno, A. Choudhary, B. Narahari, N. Memon, and R. Simha. SAFE-OPS: A Compiler/Architecture Approach to Embedded Software Security, *ACM Transactions on Embedded Computing Systems*, Vol. 4, No. 1, February 2005.

[30] J. Zambreno, A. Choudhary, D. Honbo, B. Narahari, and R. Simha. High Performance Software Protection using Reconfigurable Architectures. *Proceedings of the IEEE*, Vol. 94, No. 2, February 2006.

[31] J. Zambreno, D. Nguyen, and A. Choudhary. Exploring Area/Delay Tradeoffs in an AES FPGA Implementation. *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pp. 575–585, 2004.

[32] X. Zhuang, T. Zhang, H-H. Lee, and S. Pande. Hardware Assisted Control Flow Obfuscation for Embedded Processors. *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, September 2004

[33] X. Zhuang, T. Zhang, and S. Pande. HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus. *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2004.