A Simple Compiler-FPGA Technique to Detect Memory Spoofing in Encrypted-Execution Platforms

Eugen Leontie*, Olga Gelbart, Bhagirath Narahari, Rahul Simha
Department of Computer Science
The George Washington University
Washington, DC 20052

February 7, 2007

Abstract

In an Encrypted Execution and Data (EED) platform, instructions and data are stored in encrypted form in memory which the processor decrypts when fetched and re-encrypts for stores to memory. EED platforms, while acknowledging the overhead of decryption or re-encryption, have proven to be an attractive because they offer strong security against tampering and information leakage. Nonetheless, several attacks are possible even with EED platforms. This paper presents an approach to address a class of such attacks that we term *memory spoofing*, in which an attacker is assumed sophisticated enough to control the address bus and spoof memory blocks as they are loaded into the processor. The approach presented makes use of accompanying FGPA hardware, something that is now commonly available on many processor chips, and exploits cache boundaries to simplify integrity-checking. An additional advantage of this approach is that all the EED primitives are implemented in the FPGA and therefore the entire combination of CPU, cache and memory controller is left untouched. Experimental results using the MiBench in SimpleScalar show an average of 6% overhead using this approach.

1 Introduction

We focus on a class of attacks that we term *memory spoofing* aimed at Encrypted Execution and Data (EED) platforms. EED platforms are typically designed for attackers who use their access to the address and data buses to sniff for information (intellectual property) or to manipulate memory and execution directly by controlling the bus. EED platforms, inspite of their overhead, are especially attractive in embedded systems because small devices are physically in the hands of an attacker, who might have probes capable of sniffing or controlling the bus. Nonetheless, as we argue, a sophisticated attacker using modern electronic laboratory equipment can mount several types of memory-spoofing attacks on EED platforms. These do not reveal information but control the flow of execution, which in turn may allow an attacker to circumvent license checks or other provide access to unauthorized features.

In this paper, we describe an approach to detecting three types of memory-spoofing attacks on EED platforms. In the most elementary form of this attack, an attacker controls the bus, waiting for the processor to fetch a memory block, and then supplies the wrong (but properly encrypted) memory block; thus, the attacker, instead of decrypting, merely plays with the already encrypted blocks. We classify such attacks into three types: one in which seeks to disrupt execution by supplying a block with random content, one in which an attacker "replays" a prior block (that is therefore correctly encrypted) and a more sophisticated one in which control-flow is hijacked.

*Contact author: eugen@gwu.edu. This work is partially supported by NSF Grant ITR-0325207 and AFOSR grant FA9550-06-1-0152

To address this problem, we instrument the back-end of the compiler and propose the use of additional hardware in the processor chip. We assume that compilation itself occurs in a safe location and that the additional hardware cannot be manipulated by the attacker since it is inside a chip. For the hardware, we opt for FPGA (Field-Programmable Gate Array) technology that is now commonly available on several processor chips. The technique works as follows. First, the back-end compiler module instruments the executable so that each cache block has a special label containing the start address of the block. Second, the FPGA module, which we will call the *guard*, intercepts cache block requests from the memory controller, and processes each encrypted cache block, checks against memory-spoofing and passes on the decrypted cache block to the processor. It is this module, as we describe in detail later, that uses the compiler-inserted labels to detect spoofing.

The core contribution of this paper is the technique itself: the contents of the cache-block labels, the manner by which the labels are used in integrity-checking, and its efficiency: an average of less than 6% overhead on compute-intensive benchmarks. Our approach has several positive features and we acknowledge, one disadvantage. One attractive feature is that a single piece of information (in the label) is used to detect all three types of memory-spoofing attacks. A second advantage is that the labels are easily inserted post-compilation and, therefore, our approach can be applied to legacy binaries. A third arises from the use of FPGA's: we both show how a basic EED platform can be implemented using FPGA hardware, leaving the standard processor components unmodified, and how the FPGA can be used to optimize the computations involved in decryption and integrity-checking. Furthermore, because the FPGA is reprogrammable, encryption algorithms can be changed post-deployment. The industry also pays attention to providing FPGA logic with resistance to physical attacks [26] Our approach leaves intact the other protections offered by EED platforms, against information leakage and code tampering. There, however, is one disadvantage: our approach requires knowledge of the cache block size and the address where the program is loaded, because address offsets are part of the labels. This is not so much a problem in embedded systems where, typically, this information is known prior to deployment. However, it may require a special secure loader **cite Arbaugh paper** for large servers or for desktop computers.

We also point out that our approach and EED platforms in general are not aimed at higher-level attacks resulting from, say, buffer overflows or known vulnerabilities in operating systems. Instead, EED platforms are expressly targeted at bus-sniffing or direct probing of memory, and complement protections for higher-level attacks.

The rest of the paper is organized as follows: Section 2 discusses previous work; Section 4 depicts possible attacks on encrypted execution and the details of our approach; Section 5 provides an analysis of our approach, focusing on security and performance optimization; Section 7 presents the experimental results, after which concluding remarks are given in Section 8.

2 Related work

The general area of computer security, and in particular, software protection, has grown tremendously over the past decade. Thus, even in our own niche of compiler or hardware-based approaches, there is now a significant literature that includes overview and survey articles [5, 6, 8, 17, 41, 45]. We will thus restrict ourselves to reviewing related work in compiler-hardware approaches, and in FPGA-related work in the area of security.

Hardware approaches can be categorized into co-processor solutions [38, 43, 46, 42, 22], smartcard applications [25] (which is a type of co-processor solution), solutions that specify particular architectures or use FPGA's. FPGA's have been used to implement accelerated versions of several well-known cryptographic primitives such as private-key algorithms [15, 21, 23, 24, 39], public-key algorithms [16, 31, 36], and secure hash algorithms [19, 30]). Much of the recent work in this area has focused on implementing high-throughput or low-area Symmetric key Block Cipher (SBC) architectures on FPGAs [49, 50]. Examples include the Data Encryption Standard (DES) [34], the Advanced Encryption Standard (AES) [33], the International Data Encryption Algorithm (IDEA) [28], the Serpent [3] block cipher, and the Twofish [40] encryption algorithm.

Among architectures specifically designed for software protection, there is past work that on memory

protection [7, 44], on specific attacks [37], or even the initialization of a system [1]. Our own work in this area [47, 48] 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. Among the first of these is the XOM architecture [29] 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. Among these are our own work [] and the work of Pande et al [52, 53]. In [52, 53], 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 [18] has focused on control-flow attacks. This paper presents an alternative approach that is based on exploting cache-block boundaries.

3 Attacks on EED Platforms

Before describing our approach, we review several types of attacks on EED platforms that together constitute the attack model for our approach. To begin, let us first consider the basic elements of an EED platform: executables are encrypted and remain in encrypted form in memory; when instructions or data are fetched to the processor (across the untrusted bus), they are decrypted inside the processor, which is assumed to be trusted. Likewise, when the CPU writes data back to memory, the data is encrypted and then transmitted across the bus to memory.

At first glance, one assumes that a sufficiently strong key will completely protect execution. After all, if the key can't be broken, no information is lost and no attacker can insert their own code. However, encryption is performed in blocks because it is prohibitively time-consuming or impossible (because the cache may not be able to hold the entire program) to decrypt the entire program at once. Thus, encryption is organized around smaller blocks that are individually decrypted as and when needed. Similarly, data blocks when written are encrypted in small blocks for the same reason, efficiency.

The fact that encryption occurs in blocks enables a sophisticated attacker to mount some attacks on EED platforms, as we outline below. Such an attacker will be able to not only sniff the bus but to actively control it. 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:

- *Execution Disruptions*: In this attack, 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 deciphered, 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*, because the attacker is not really able to insert precisely engineered code, but is able to perturb normal execution without detection, which in turn can lead to

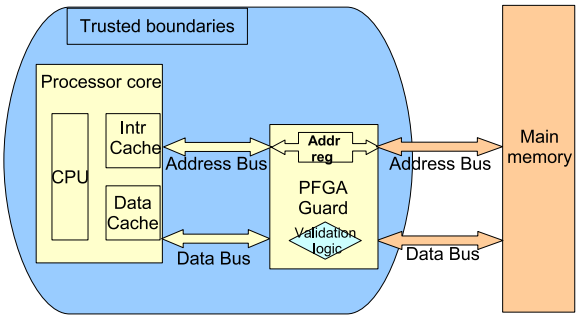

Figure 1: Architecture: the FPGA Guard

other attacks. Moreover, by observing the processor, the attacker can infer to some extent how known bit sequences are decrypted into instructions, thus providing excellent cribs by which the encryption itself can be attacked.

- *Replay Attacks.* 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. 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, as many times as the action needs to be triggered, without the attacker having to guess the entire instruction block functionality.

- *Control Flow attacks.* As described in [52], an attacker can observe patterns on the bus to infer the control-flow structure of the executable. This allows a so-called control-flow attack in which a cryptographically valid, but control-flow invalid, block is supplied to the processor. There are two types of control flow attacks that we distinguish. Consider three blocks A, B and C and suppose that in normal execution, block A can transfer control to either block B or block C. An attacker can substitute C when B is requested and observe the outcome as a prelude to further attack. The second 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.

Taken together, the 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 such attacks.

4 System Description and Approach

Our approach is currently designed for a standard Harvard architecture (with separate instruction and data memory) and 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 backend compiler module that instruments the executable such that each cache block has a label. The third is a detection algorithm that examines the labels of cache blocks to verify proper execution. This paper focuses exclusively on protecting instruction memory; data memory issues, which are similar in some ways but different in others, will be addressed in a forthcoming paper.

Figure 4 shows a processor chip on the left and main memory on the right. We assume that the chip comes with FPGA logic, as do many commercial processors today. We use this logic to implement the *guard*

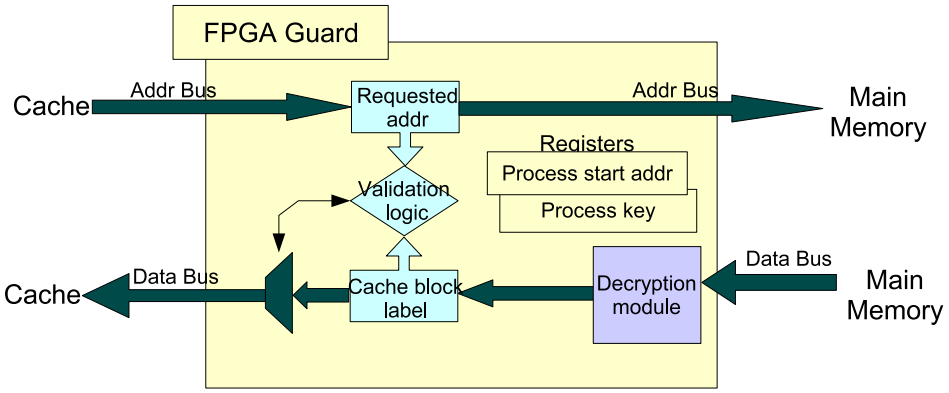

Figure 2: FPGA Guard – detailed view

functionality we need to verify memory accesses. To see how this works, consider how memory accesses take place without such guard logic: when a cache miss occurs, the memory management logic 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 guard. The guard logic is then aware of the start address of an instruction cache 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 then able to perform decryption and examine the contents of each instruction cache block before it is fed into the instruction cache. And that is the key to ensuring trust: the blocks that reach the cache have been verified by the guard so that the processor sees (and therefore executes) only validated blocks.

Next, we focus on what the guard examines in each decrypted cache block – see Figure 4; the decryption itself is straightforward and has been studied in many of the papers cited in Section 2. As part of post-compilation, each block of instructions has a label inserted into the block; the label itself is stripped from the block and is not passed into the cache. Each label consists of two pieces of information: the offset of this block from the base address of the executable, and an integrity-checksum of the block. Thus, the guard is able to examine whether the block is actually the block corresponding to the memory access that was requested, and whether the block has been tampered with.

We will use this code snippet in assembly, assumed to be part of a single cache block, to illustrate the above ideas:

```
1. MOV   R0, #3          ; Load a bit mask (2'b11) into R0
2. ADD   R1, R2, #100    ; ADD the content of R2 with 100 and put in R1
3. CMP   R2, #3          ; Compare R2 with '3'
4. BEQ   100(R1)         ; branch on equal to 100+R1
5. CMP   R3, #2          ; Compare R3 with '2'
6. BEQ   0x010C          ; branch on equal to 0x010C
```

We see that after executing the instructions in lines 1-3, the processor executes the branch instruction at line 4. Depending on the outcome of the comparison, the program counter is loaded with either `100+R1` or `PC+4` (where PC is the program counter). If this target address is not in the cache, the cache controller issues a read to memory, during which the guard captures the read address and issues the remaining cache-block memory read operations. The guard then decrypts, checks integrity and label correctness and delivers the cache block as needed by the cache controller. The label check compares the label+base to the address captured by the FPGA. If, on the other hand, the target address is already in the cache, it was decrypted and checked earlier and is hence safe.

The following summarizes the roles of the compiler and the FPGA guard logic:

- **Compiler role**:
  - Divide the code into blocks the size of a cache block.
  - Reserve space in each block for the label information. This involves re-computing branch labels.
  - Compute each block's label: the relative address of the first instruction in each block, and integrity checksum.
  - Encrypt the cache block using a unique key for the program.
- **FPGA guard role:**
  - Each time there is a miss in cache memory, the cache controller makes a request to main memory to fetch the block.
  - The guard intercepts the first address – 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 label in the block, and checks if label offset + base address is equal to the snooped address.
  - If the validation succeeds, the guard continues by checking integrity.
  - 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.
  - Assuming integrity succeeds, the guard feeds the decrypted cache block into the processor replacing the label with NOP instructions.

5 Attack Analysis

We first explain how our simple mechanism detects the three types of attacks described earlier before discussing, in the following section, the validity of the assumptions made in our approach. First, note that execution disruptions are detected using the the message-digest, as in any block-based EED platform. Second, replay and control-flow attacks are detected because the guard always returns the correct block to the cache; in other words, when the cache controller requests an block (by providing the start address), the label-checking mechanism assures that no other block is delivered to the cache.

To see how this works, consider first a control-flow attack. Suppose block A transfers control to either block B or block C, depending on runtime conditions, and that block A initially generates the address for (the start of) 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 will conflict with the block C address that the guard captures, thus allowing the guard to detect the substitution. 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? Here, it is important to distinguish between the buffer-overflow event (caused by a programmer error, say) and the standard stack-smashing attack that seeks to inject code. Because buffer-overflows are considered a language "feature" in an EED platform, neither the standard encryption nor our labeling mechanism checks against array boundaries. However, any code injection is caught because the code injected would have to be properly encrypted. 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 labeling mechanism.

6 Discussion of Assumptions

There are several points worth clarifying at this time, starting with the post-compilation pass over the executable. First, 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, the label generation and branch target modification can easily be done in a single pass since the cache block size and label size is fixed, and can therefore be used compute the new branch targets. Fourth, we have not explicitly provided the details of computing the integrity checksum, nor the encryption itself, because both are relatively straightforward and, being the cornerstone of EED platforms, have been addressed elsewhere.

Fifth, turning to the guard, 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. We explore this issue further in the section on experimental results. Also, because the guard is implemented in FPGA logic, a variety of optimizations can be introduced to perform decryption in parallel with integrity checking, an issue we discuss later in this paper.

Next, we note that the guard needs to know the base address for a program. 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.

Note that we have also assumed that the cache size is known at compile time. This is not an unreasonable assumption in many applications, but it does reduce portability of the encrypted executables. Cache sizes usually range from 8 to 512 bytes, typical of processors such as ARM, PowerPC, Microblaze, or OpenRISC. Although our current experiments were designed for a 32 bit architecture, the model is easily extended for a 64 bit processor with relative small changes.

Finally, to accelerate the validation in the guard, we considered the possibility of using more than one AES decryption engine that decrypt in parallel. For a 32 byte cache block, for example, we could use two 16-byte AES decryption blocks. But this speed-up approach raises another issue. Instead of the serial AES in cipher block chaining (CBC) mode [13], we need to use the Electronic CodeBook (ECB). Each 128 bit blocks is individually encrypted. This allows individual blocks to be replaced. For this reason, before encryption, each block goes through a permutation operation that takes 50% of the label bits and places in the second block. The guard performs the inverse permutation after decrypting the data. For 64 byte cache blocks, each AES block contains 25% of the label. Even if the speedup gained could be invaluable when considering real-time and speed optimized systems, the price payed is higher area taken by the FPGA logic and a weaker protection. Although the chance of producing a valid cache block remains $1 \setminus 2^{32}$, the attacker can target only one AES block, with $1 \setminus 2^{16}$ chances of successfully disrupting the execution..

7 Experimental Results

For the overall simulation of our system, we used the SimpleScalar simulation suite [2] for an ARM processor architecture [4]. 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 analysis was performed on 32-byte line and 64-byte line caches, since cache block size has the most impact on the system performance. The rest of the simulation

parameters are synthesized in the table 7. The simulator used is sim-outorder.

Parameter Name	Parameter Value	Parameter Name	Parameter Value
bpred	bimod	decode:width	4
issue:width	4	issue:inorder	false
issue:wrongpath	true	commit:width	4
cache:dl1	dl1:16:32:64:l	cache:dl1lat	1
cache:dl2	none	cache:dl2lat	1
cache:il1	il1:32:32:32:l	cache:il1lat	1
cache:il2	none	cache:il2lat	1
cache:flush	false	cache:icompress	false
mem:lat	(depends on cache size)	mem:width	8
mem:pipelined	false	res:ialu	4
res:imult	1	res:memport	1
res:fpalu	4	res:fpmult	1

Table 1: SimpleScalar parameters

The benchmarks chosen for the simulations were computational intensive applications from MiBench [20] : bitcount - tests the bit manipulation abilities of a processor by counting the number of bits in an array of integers; crc - checksum calculation for a file; dijkstra - an implementation of the graph algorithm for calculating the shortest paths between nodes; fft - Fourier transforms are used in digital signal processing to find the frequencies contained in a given input signal; sha - the standard secure hashing algorithm used in most security transactions; stringsearch search algorithm for given words in phrases using a case insensitive comparison algorithm; susan - an image processing suite - with three variants : corners , edges, smoothing. Field, Pointer, Transitive and Update are data intensive benchmarks.

Since the technique used operates at cache block level, the overhead incurred by the encryption and the validation mechanism affect each cache block fetch from main memory. Whenever a cache miss occurs,the delays by the Guard's operations are added to the access time to the lower level memory . The overall performance penalty is affected by three factors: increased cache misses, extra instruction executions 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 instruction fit in the same cache memory space. The average increase in cache miss rate is 19.18% for the 32 byte cache blocks. A detailed cache miss graph for each benchmark is depicted in Figure **??**.

The second penalty source is the execution of the extra nop instruction in each cache block. The encryption and validation adds a fixed penalty for each memory fetch for each instruction cache block. The operations performed by the Guard can be modeled as an increased latency in the instruction fetch. Figure 7 depicts

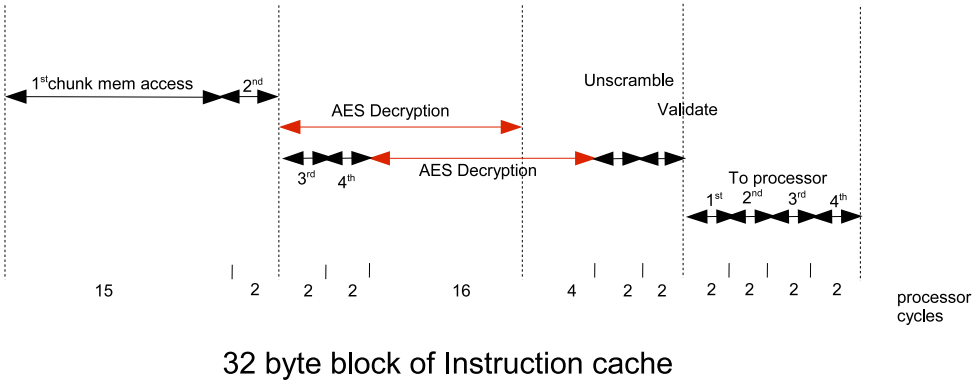

Figure 3: FPGA Guard Details

how the penalty cycles were estimated for a 32byte cache block. In the architecture considered, the processor speed is 200Mhz and the FPGA half its speed (100Mhz), so every FPGA computation cycles that does not overlap processor execution creates 2 processor penalty cycles. Recent FPGA implementations of AES manage to achieve high throughput by pipelining the execution path and unrolling techniques [33]. The AES decryption implementation chosen by our model is one that minimizes the decryption penalty since high throughput is not the target in this architecture. The 10 FPGA cycles for the decryption translates into 20 processor penalty cycles that are added to the cache miss penalty. The other Guard processing are : 1 cycle for address validation and one cycle for the inverse permutation of the cache block bits. Cache miss penalty is computed according to the equation :

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

Benchmark	Cache miss penalty(%)	Added Penalty by nop(%)	Overall Penalty(%)
Bitcount	18.76	6.43	6.47
Crc	17.1	12	12.03
Dijkstra	18.98	2.71	2.74
Fft_inv	22.81	3.45	3.7
Fft	23.21	1.65	2.78
Sha	16.05	-1.07	-0.96
Stringsearch	18.6	-0.66	6.87
susan.corners	18.29	3.92	6.35
Susan.edges	21.69	2.25	3.64
susan.smoothing	17.62	3.46	3.53
Field	17.29	10.03	10.05
Pointer	16.67	0.17	0.19
Transitive	16.06	-3.13	4.13
Update	16.78	2.07	14.41
Average	18.56	3.16	5.42

Table 2: Performance penalty parameters for 32 byte cache blocks

The baseline to which the penalties are computed is a standard program execution with no encryption or any other security method. Tables 7 and 7 show the performance of the system with details on how much the extra nop, inserted in the instruction stream, affects the execution, and how much penalty comes from the extra encryption and validation in the Guard. As expected the larger cache blocks incur less overhead, since the ratio of nop:original instructions is larger. On average the extra penalty caused by the nop insertion is 3.51% and the added validations accounts for a total of 4.71%. The performance penalty is even less for the system with 64 bytes cache blocks:2.12% from the nop execution and 2.91% total. The increase in program size is also a major concern in embedded applications. Using only 32 bits for keeping the integrity validation data, the protection scheme that we are proposing increases the overall program size by only 12.5% for the 32 byte cache blocks and 6.25% for the 64byte ones.

The last table (7) compares the overhead of the protection mechanism for both 32 and 64 byte cache blocks with a baseline consisting in EED execution. The results show that the size of the cache block has a major influence on the performance of the benchmarks. Since the major performance penalty comes from the execution of the extra nop and the cache misses that it causes, the larger cache block has a smaller ratio of overhead code per workload code and though achieves better results. For all the benchmarks analyzed, the miss rate for the instruction cache is very small - 0.001% on average - and so the added penalty for the encryption is significantly lower than the overhead of the nop. This observation motivates further analysis on how to eliminate passing the extra nop to the processor, modifying the guard to act as an additional upper level cache controller. The space utilization by the fpga guard is very low, the major component being the aes-decryption (284 slices and 7 BRAM for a synthesis on XUPXilinxVirtex2Pro) and with minimal on-chip memory (current address buffer and process keys).

Benchmark	Cache miss penalty(%)	Added Penalty by nop(%)	Overall Penalty(%)
Bitcount	13.02	3.99	4.02
Crc	13.5	2	2.03
Dijkstra	11.89	2.15	2.18
Fft_inv	17	2.47	2.62
Fft	16.85	2.55	2.64
Sha	10.82	4.22	4.29
Stringsearch	12.72	-1.44	3.57
susan.corners	12.68	3.33	4.87
Susan.edges	15.42	1.98	2.88
susan.smoothing	12.72	-0.01	0.04
Field	13.31	9.24	9.27
Pointer	14.1	-0.3	-0.28
Transitive	12.01	3.53	10.53
Update	11.81	2.44	14.26
Average	13.42	2.58	4.49

Table 3: Performance penalty parameters for 64 byte cache blocks

8 Conclusions and Future Work

This paper proposed a powerful method of code protection from physical attacks and stops most of the buffer overflows attacks. Our work continues with a scheme to use the FGPA Guard in an efficient way to protect data located in the untrusted memory and on techniques to reduce the cache miss rates. An actual implementation on the system on a Virtex2 FPGA Testing platform from Xilinx is also the target of our current work.

9 Acknowledgments

Authors want to thank Stefan Popoveniuc for preliminary discussion and analysis on EED attacks.

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

[3] E. Biham, R. Anderson, and L. Knudsen. Serpent: A New Block Cipher Proposal. *Proceedings of the International Workshop on Fast Software Encryption (FSE)*, pp. 222–238, 1998.

[4] D. Brash. *The arm architecture version 6*, ARM Whitepaper available at www.arm.com (January 2002).

[5] S. Cheng, P. Litva, and A. Main. Trusting DRM software. *Proceedings of the Workshop on Digital Rights Management for the Web*, January 2001.

[6] C. Collberg, C. Thomborson and D. Low, *A taxonomy of obfuscating transformations*, Technical Report 148, Department of Computer Science, University of Auckland (July 1997).

[7] 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.

Benchmark	32byte(%)	64byte(%)
Bitcount	6.44	3.99
Crc	12	2.01
Dijkstra	3.2	2.74
Fft_inv	7.18	6.95
Fft	4.7	5.01
Sha	-1.04	4.23
Stringsearch	2.81	1.78
susan.corners	6.78	6.41
Susan.edges	4.08	3.93
susan.smoothing	3.52	0.06
Field	10.02	9.23
Pointer	0.18	-0.31
Transitive	0.05	2.33
Update	5.24	0.51
Average	4.65	3.49

Table 4: Performance penalty compared to EED

[8] C. Cowan. Software Security for Open Source Systems. *IEEE Security and Privacy Magazine*, Vol. 1, No. 1, pp. 35–48, February 2003.

[9] 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).

[10] Dallas Semiconductor, Inc. Features, Advantages, and Benefits of Button-based Security. Available at `www.ibutton.com`, 1999.

[11] A. Dandalis, V. Prasanna, and J. Rolim. An Adaptive Cryptographic Engine for IPSec Architectures. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 132–144, 2000.

[12] A. Das, G. Memik, D. Nguyen, J. Zambreno, and A.Choudhary. An FPGA-based Network Intrusion Detection Architecture. *IEEE Transactions on Information Forensics and Security* (to appear), 2007.

[13] M. Dworkin, *Recommendation for block cipher modes of operation*, NIST Special Publication 800-38A (2001 Edition).

[14] J. Dyer, M. Lindemann, R. Perez, R. Sailer, S. Smith, L. van Doorn, and S. Weingart. Building the IBM 4758 Secure Coprocessor. *IEEE Computer*, Vol. 34, pp. 570-66, October 2001.

[15] 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.

[16] M. Ernst, M. Jung, F. Madlener, S. Huss, and R. Blumel. A Reconfigurable System on Chip Implementation for Elliptic Curve Cryptography over GF(2n). *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pp. 381–399, 2002.

[17] M. Fisher. Protecting Binary Executables. *Embedded Systems Programming*, Vol. 13, No. 2, February 2000.

[18] O. Gelbart, P. Ott, B. Narahari, R. Simha, A. Choudhary, and J. Zambreno. CODESSEAL: A Compiler/FPGA Approach to Secure Applications, *Proceedings of the IEEE Conference on Intelligence and Security Informatics (ISI)*, 2005.

[19] T. Grembowski, R. Lien, K. Gaj, N. Nguyen, P. Bellows, J. Flidr, T. Lehman, and B. Schott. Comparative Analysis of the Hardware Implementations of Hash Functions SHA-1 and SHA-512. *Proceedings of the International Conference on Information Security (ISC)*, pp. 75–89, 2002.

[20] 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).

[21] 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.

[22] N. Itoi. Secure Coprocessor Integration with Kerberos V5. IBM Research Report RC-21797, IBM TJ Watson Research Center, July 2000.

[23] K. U. Jarvinen, M. T. Tommiska, and J. O. Skytta. A Fully Pipelined Memoryless 17.8 Gbps AES-128 Encryptor. *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pp. 207–215, 2003.

[24] J.-P. Kaps and C. Paar. Fast DES Implementation for FPGAs and its Application to a Universal Key-Search Machine. *Proceedings of the Annual Workshop on Selected Areas in Cryptography (SAC)*, pp. 234–247, 1998.

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

[26] J. Kumagai, *Chip detectives [reverse engineering]*, Spectrum, IEEE (Nov 2000).

[27] J. Lach, W. Mangione-Smith, and M. Potkonjak. FPGA Fingerprinting Techniques for Protecting Intellectual Property. *Proceedings of the IEEE Custom Integrated Circuit Conference*, pp. 299–302, May 1998.

[28] X. Lai and J. Massey. A Proposal for a New Block Encryption Standard. *Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques*, pp. 389–404, 1990.

[29] 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).

[30] R. Lien, T. Grembowski, and K. Gaj. A 1 Gbit/s Partially Unrolled Architecture of Hash Functions SHA-1 and SHA-512. *Proceedings of the Cryptographers' Track at the RSA Conference (CT-RSA)*, pp. 324–338, 2004.

[31] 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.

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

[33] 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.

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

[35] Alef One, *Smashing the stack for fun and profit*, Phrack, vol.7, no. 49, Nov. 1996.

[36] S. Okada, N. Torii, K. Itoh, and M. Takenaka. Implementation of Elliptic Curve Cryptographic Coprocessor over GF(2m) on an FPGA. *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pp. 25–40, 2000.

[37] 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).

[38] E. Palmer. An Introduction to Citadel—A Secure Crypto Coprocessor for Workstations. Research Report RC 18373, IBM T.J. Watson Research Center, 1992.

[39] G. Saggese, A. Mazzeo, N. Mazzoca, and A. Strollo. An FPGA-Based Performance Analysis of the Unrolling, Tiling, and Pipelining of the AES Algorithm. *Proceedings of the International Conference on Field-Programmable Logic and its Applications (FPL)*, pp. 292–302, 2003.

[40] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson. *The Twofish Encryption Algorithm: A 128-Bit Block Cipher*. John Wiley and Sons, 1999.

[41] R. Simha, A. Choudhary, B. Narahari, and J. Zambreno. An Overview of Security-Driven Compilation. *Proceedings of the Workshop on New Horizons in Compiler Analysis*, December 2004.

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

[43] J. Tygar and B. Yee. Dyad: A System for Using Physically Secure Coprocessors. *Proceedings of the Harvard-MIT Workshop on Protection of Intellectual Property*, April 1993.

[44] 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.

[45] J. Wyant. Establishing Security Requirements for More Effective and Scalable DRM Solutions. *Proceedings of the Workshop on Digital Rights Management for the Web*, January 2001.

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

[47] 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.

[48] 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.

[49] 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.

[50] J. Zambreno, D. Honbo, and A. Choudhary. Exploiting Multi-Grained Parallelism in Reconfigurable SBC Architectures. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 333–334, 2005.

[51] J Zambreno, T Anish, and A Choudhary, *A run-time reconfigurable architecture for embedded program flow verification*, Proceedings of the NATO Advanced Research Workshop (ARW) on Security and Embedded Systems (2005).

[52] 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

[53] 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.