Architectural support for Securing Application Data in Embedded Systems

Olga Gelbart,Eugen Leontie, Bhagirath Narahari, Rahul Simha
Department of Computer Science
The George Washington University
Washington, DC 20052
Contact: eugen@gwu.edu

***Abstract*—The rapid growth and pervasive use of embedded systems makes it easier for an adversary to gain physical access to these devices to launch attacks and reverse engineer of the system. Encrypted execution and data (EED) platforms, where instructions and data are stored in encrypted form in memory, while incurring overheads of encryption have proven to be attractive because they offer strong security against information leakage and tampering. However, several attacks are still possible on EED systems when the adversary gains physical access to the system. In this paper we present an architectural approach to address a class of memory spoofing attacks, in which an attacker can control the address bus and spoof memory blocks as they are loaded into the processor. In this paper we focus on the integrity of the application data to prevent the attacker from tampering, injecting or replaying the data. We make use of an on-chip FPGA, an architecture that is now commonly available on many processor chips, to build a secure on-chip hardware component that verifies the integrity of application data at run-time. By implementing all our security primitives on the FPGA we do not require changes to the processor architecture. We present that data protection techniques and a performance analysis is provided through a simulation on a number of bechmarks. Our experimental results show that a high level of security can be achieved with low performance overhead.**

I. Introduction

Embedded systems are especially vulnerable to attacks since they 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. In this paper we focus on a class of attacks that we term *memory spoofing* aimed at Encrypted Execution and Data (EED) platforms and embedded systems. 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 seem especially attractive in embedded systems that are susceptible to physical capture. 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 can disrupt execution and alter the control flow of the program.

In this paper, we describe an architectural approach to detecting and preventing three types of memory-spoofing attacks on the application data in 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: an injection attack in which they seek to disrupt execution by supplying a data block with random content, a substitution attack in which an attacker substitutes a correctly encrypted block from elsewhere in memory, and a stale data replay attack in which the attacker substitutes a correctly encrypted, with the requested address, but stale data block.

Our approach can be summarized as follows. The hardware platform we target is a standard processor with an accompanying on-chip reconfigurable logic core in the form of an FPGA – this enables us to leverage commercially available System-On-Chip (SOC) architectures, such as chips from the Xilinx family. 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 read and write 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. When application data is generated and written to memory, the guard encrypts the data while also embedding sufficient additional integrity checking information to detect attacks. It is this guard module, as we describe in detail later, that uses the embedded integrity checking information to detect spoofing. While this paper focuses on securing the application data, our techniques complement our earlier work on securing the application code [15] and thus taken together provide a complete system to protect against both code and data tampering attacks.

The contribution of the paper is the approach itself: 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 protections of the application code and data without changing the architecture. The architecture and design of the secure hardware component is motivated by emerging trends in em-

bedded systems: standard processor cores are augmented with FPGA fabric [18]. Since many embedded systems typically have timing requirements, the overhead incurred determines the feasibility of a protection scheme. The low overheads incurred by our system, as tested on various benchmarks from the MiBench suite of benchmarks, validate the feasibility of our solution. Our approach differs from prior work in that we focus on ease-of-adoption – our approach does not require processor re-design because we use on-chip reconfigurable logic in the form of a Field Programmable Gate Arrays (FPGA). Since all our security primitives are implemented on the FPGA, the entire combination of CPU (instruction set and microarchicture), cache organization and main memory is left untouched. In addition, because our techniques are directly incorporated into the compiler backend and the hardware our approach imposes no burden on software developers. We

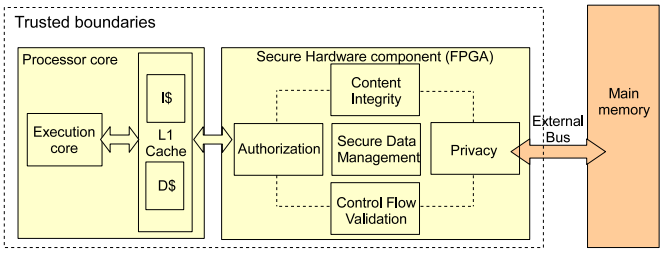

Fig. 1. Codeseal architecture

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. Our protection mechanism does not protect against code vulnerabilities such as buffer overflows. Also, although an FPGA constant power consumption would obfuscate the CPU power drain, our approach does not address power analysis attacks

The rest of the paper is organized as follows: Section II discusses previous work; Section IV discusses our threat model – EED attacks after physical capture of the system. Section IV presents a conceptual view of our architecture; Section V discusses our implementation and a security analysis; Section VI presents the experimental results, after which concluding remarks are given in Section VII.

II. Related work

Software only approaches, for code or data security, will not prevent against attacks under our threat model where the adversary has physically captured the device. Therefore we restrict ourselves to reviewing related work in architectural support for software security.

Architectural solutions specifically designed for protecting specific software vulnerabilities, such as buffer overflow, include the past work of [3] or [9]. In solutions that focus on providing architectural support to protect memory, such as the Mondrian system[13] and InfoShield [14], the systems combine software fault isolation techniques together with access control enforcements to separate program memory spaces from malicious applications running simultaneously on the same system. However, the architectural changes required by these systems are extensive, involving ISA level modifications, cache hierarchy redesign and even extending the length of memory modules (external, cache or registers) with additional security information. In addition, they do not address protection against adversaries with physical access to the protected devices.

There are several projects that address the design of EED platforms, and provide both data and code integrity in a physical attack. Examples include the XOM architecture [7], and several others that tamper-resistant processors [8], [17]. Under our threat model (with a sophisticated attacker with physical access) these systems are still vulnerable to an attack. The AEGIS [16] architecture greatly improves on its predecessor(XOM) and presents techniques for control-flow protection, privacy, and prevention of data tampering. Their techniques require re-design of the processor. The project also includes an optional secure operating system kernel, which is responsible for interfacing with the secure hardware. The code and memory protection schemes employ cached hash trees and a log-hash integrity verification mechanism is used to verify integrity of memory accesses. The level of confidentiality and integrity protection is further improved by [12] in an approach to merge the two concepts of encrypted execution with access control validation.

A common theme in past work, on EED platforms, is the requirement of the design of new processors. These proposed architectural changes, to the processor and instruction set, incur a high cost in terms of design and require a buy-in from chip manufacturers and do not leverage COTS technology. These factors serve as a key motivation for our approach – we explore architectural solutions that do not require changes to the processor by exploiting commercially available platforms, such as the Xilinx Virtex Pro family, that provide a processor core with an on-chip FPGA logic. Our work in this area has explored the use of reconfigurable logic, in the form of an FPGA, towards providing software security in embedded systems without requiring a re-design of the processor . Our solutions combine hardware and compiler techniques to provide secure systems. In early work we explored embedding watermarks [10] and the use of FPGAs to provide encrypted execution [11] focusing on incurring minimum cost in terms of hardware design and performance penalty. In more recent work [15], we applied this architectural concept of using on-chip FPGA logic to present a solution for protecting code (including code injection, replay, and code modification attacks) against EED attacks. We provide code integrity and control-flow protection by using the information extracted at compile time, and the code memory layout, to enforce the correct run-time behavior of program execution. The additional hardware, in the form of an on-chip FPGA logic, ensures fast and efficient execution of the cryptographic primitives necessary for the validations. We provided solutions for known encryption execution vulnerabilities with minimum overhead by exploiting program execution properties such as code locality and security properties of the cryptographic primitives. This solution uses the same

CODESSEAL architecture model presented in this paper and, therefore, taken together with the application data protection methods presented in this paper would form a complete system that protects against EED attacks on code and data.

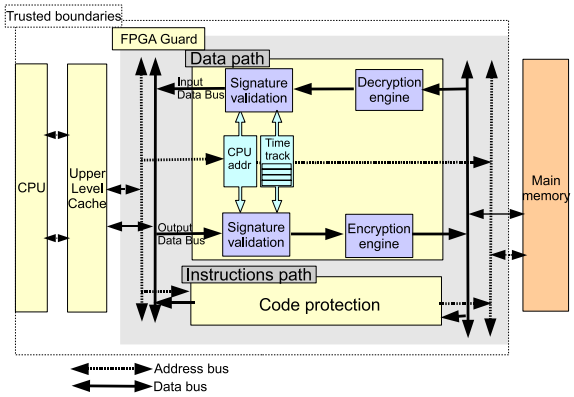

Fig. 2. Data And Instructions Paths

III. Data Attacks on EED Platforms

Before describing our approach, we discuss our threat model and the types of attacks that are possible on EED platforms under physical capture. Since protection of application data is the focus of this paper, we direct the reader to [15] for a discussion of EED attacks on code and program control flow. In an EED platform both code and data are encrypted and remain in encrypted form in memory; when the memory content is requested by the CPU in the execution process (across the untrusted bus), the memory blocks are decrypted inside the execution unit. We assume that the single chip execution unit forms our trusted boundary. Likewise, when the CPU writes data back to memory, the data is encrypted and then transmitted across the bus to memory.

Encryption, of application data or code, is typically 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. A sophisticated attacker can exploit this by manipulating data and address lines and change the encrypted blocks without the decryption mechanism noticing any difference. While instruction code blocks are harder to forge, since not any code block will decrypt into valid execution codes, data blocks do not have any restriction and they gain meaning only in the context of a software application. Although hard to replicate a forged exact value inside an encrypted block without knowing the encryption key, brute force can lead to approximate values in reasonable times, thus tricking the running application and changing its behavior. We describe the three types of application data attacks that are considered in this paper – data injection, data substitution and stale data replay.

Data Injection. 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 since they do not know the encryption key). 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 data. The attacker may still be able to disrupt the execution or learn something about the executable. In addition, 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.

Data Substitution. The attacker can substitute a currently-requested data block for another data block generated by the application, and therefore correctly encrypted, and thus observe the program's behavior. For example, if the program requests data from address A the attacker can inject data from address B. Once again, simply encrypting the data does not prevent such data injection attacks.

Stale Data Replay. Data is even more sensitive to replays than code is as its content changes in time. If we consider a block of data and its evolution in time as the program executes, time 0 being the loading time, we can observe why. Any change in the content of the block creates a new valid encryption of that block in memory. An attacker that sniffs the bus can keep snapshots of data (ex. if time 1, 2, 3 represent write-backs to the same data block the attacker has 4 valid encryption of that block, including that of the initial state). If the block holds security session information, the attacker could use block 2,for example,when the processor requests it without triggering any exception from the validating hardware and driving the program into an expired state.

Taken together, the attacks point out that mere encryption is not sufficient to guarantee proper execution and data protection and that these types of attacks can go undetected unless we provide explicit support. What is required, as also observed by others [16], is some form of integrity checking information that needs to be associated with the application data. We now turn to our approach in which we insert integrity information into the data blocks, and utilize FPGA logic on the chip to implement the validation logic required. We note that the compiler inserts the integrity information for static data and the hardware inserts this information for dynamic data.

IV. Our Approach and the Architecture

Our secure execution tool chain [4] relies on two main components: a front-end tool that generates a security-annotated binary and a hardware mechanism that implements the algorithm for intrusion detection and prevention. Designed as a general mechanism for protecting both control-flow and data privacy our system achieves high security standards with relative modest performance penalties. The architecture and design of the hardware validation component is motivated by emerging trends in embedded systems: standard processor cores are augmented with FPGA fabric [18]. We build our validation logic in the FPGA for the following reasons: no change is needed to processing core and the validation mechanisms are easily configurable to allow new validation mechanism. Overall the costs of building such systems are

greatly reduced this way. The compiler is responsible for code layout analysis and provides the information needed by the code generation step (the linker) on the structure of memory regions. It is the last stage of the compilation that performs the encryption and signature computations. Focusing on data privacy and integrity protection, the compiler's analysis derives three main code regions: static data (constants, initialization values, string tables) run-time data (stack and heap) and shared data (memory regions that do not hold sensitive information and/or are used for inter-process communication). Run-time

Fig. 3. Signature Memory Layout

data has the property that its lifespan lasts only while the process is active. The keys needed for their encryption do not leave the guard premises. On the other side, static data is encrypted by the compiler, thus the key information needs to be brought into the guard. The complexity of the protected system dictates the mechanism for key establishment. The key is either preloaded (in the most simple form) for a dedicated device, transferred with software updates or managed by a secure kernel. For this purposes 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.As shown in fig 1 the FPGA Guard mitigates all memory communication between the CPU and the peripheral memory system. 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 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. On a eviction of a cache block to memory, the guard is responsible for re-encryption and new integrity meta-data computation.

V. Implementation and Attack Analysis

Encryption alone is a powerful mechanism to guard against eavesdropping and to enforce confidentiality. Already established as an industry standard and with fast and efficient FPGA implementations, the Advanced Encryption Standard (AES) algorithm is the perfect candidate for the encryption mechanism. But as we saw in section III it is not sufficient to enforce data integrity. A signature mechanism makes sure that the block was not modified while it is located outside the trusted bounds, the location information of the block position within the program guarantees that the no substitution attack can be performed and lastly a time-stamping mechanism stops replays of old data blocks. For the signature selection, a broad

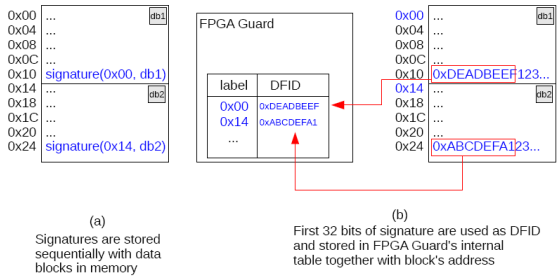

Fig. 4. Time stamping using Data Freshness Labels

range of options are available: starting with a lightweight CRC and ending up with a collision resistant secure hash mechanism such as SHA-1. The signature is either stored sequentially with the data blocks, or in a special section. In either case the FPGA guard is responsible for the address mapping translation (Fig.3). Location information is provided by the relative address of the data cache block within the program space. For fixed partitioning memory systems the address directly maps to physical addresses, where as for others the program start address provides the reference point. The label(address) information is not directly stored with the memory block. Since we already need the signature mechanism in place, the label becomes part of the signature computation. The 32 bit (16 or 64 bits depending on the processor word length) address label is concatenated with the content of the cache block and then the signature is computed. When the guard fetches the block it performs the following tasks: it fetches the cache block requested by the cache controller, decrypts its content, concatenates the start address of the block and compute the hash code of this sequence. If the computed signature matches with the hash value fetched from memory for the block the block is considered valid. An inconsistency triggers a tamper alarm. This scheme will successfully detect and prevent an attacker from injecting external data or substitute blocks from other location in memory.

To prevent timing and stale data replay a time-stamping algorithm is a required part of the integrity validation. We have not chosen a regular counter/timestamp for this purpose since counters eventually reach a maximum value, which usually requires re-encryption of the entire memory space and key re-establishment [7]. We use probabilistic model: a data freshness indicator(DFID) scheme to prevent the stale data replay attacks. As we can see in Figure 4, we use part of the signature as the data freshness indication. The DFID

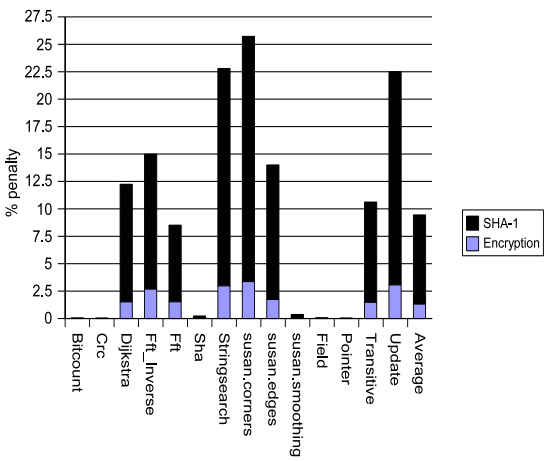

Fig. 5. Performance penalty with 32 byte cache blocks and SHA-1 signature

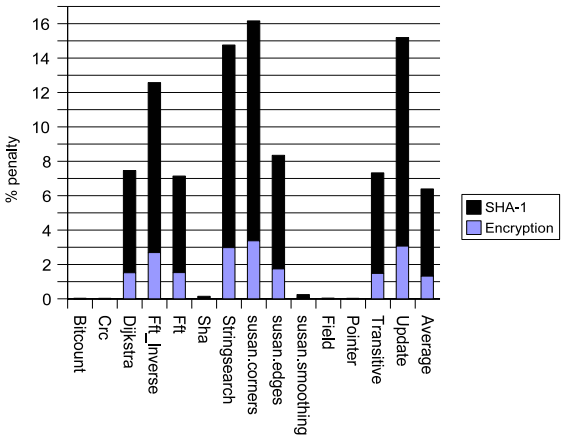

Fig. 6. Performance penalty with 64 byte cache blocks and SHA-1 signature

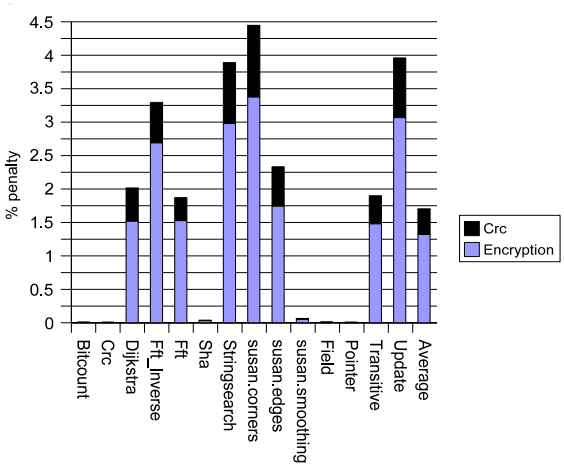

Fig. 7. Performance penalty with 32 byte cache blocks and CRC as signature

together with the data blocks address (which we also use as its label) is stored in a special table inside the FPGA Guard. Every time a data block is fetched from memory, its signature is calculated and the DFID is checked. On a dirty write, the new signature is computed and the DFID table inside the FPGA Guard is updated with the new value. We use only part of the 160-bit SHA-1 hash as the DFID (32 bits) because of storage limitations inside the Guard. The size of the DFID table inside the FPGA Guard must also be taken into account. It is reasonable to assume that for small, embedded applications there can be enough storage space for all DFIDs needed. However, in a general case, the FPGA has to store the overflow DFIDs encrypted in memory using the same encryption/hashing/labeling/timestamping techniques for a hierarchical recursive structure. The length of the DFID can be changed to accommodate for more collision resistance or for less internal storage overhead. We can also note that DFID can be created out of any part of the encrypted data block with the same effect(we rely on the collision resistance of SHA-1 or AES in the latter).

VI. Experimental Results

In order to evaluate the performance overhead of our proposed approach, we conducted cycle accurate simulations of our architecture. In our simulation framework we made use of the SimpleScalar simulation suite [1] for an ARM processor architecture [2]. We have augmented the simulator code with an implementation of additional security module to model the role of the FPGA guard. We used the characteristics of a Xilinx Virtex-II Pro FPGA operating at 200 MHz for our FPGA implementation, and assumed an ARM processor cycle of 400Mhz. Thus every FPGA computation cycle that does not overlap processor execution creates a 2 processor penalty cycles. The external bus and main memory are assumed to run at 100 MHz. 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 and the data cache is a 32Kb , 64-way associative cache. To examine the impact of the cache block size, the analysis was performed on both a 32-byte and 64-byte line caches.

The input for evaluation were benchmarks from two different benchmark suites. We selected ten benchmarks from the computation intensive tests from MiBench embedded benchmark suite [5], and these manifest high code locality, relatively small data accesses. The cache miss rates are relatively small but represent the vast majority of algorithms for embedded devices in automotive and security domain. We then evaluated performance of four data intensive benchmarks – Field, Pointer, Transitive and Update – part of the Data Intensive Systems (DIS) benchmark suite [6]. These are a good evaluation of the extreme case applications in the embedded world. Although simpler than consumer products like PDA and mobile devices which behave more like desktop computers, the data intensive benchmarks show a high number of memory accesses, sometimes saturating the data caches and thus stressing the encryption and validation modules.

To investigate the performance impact of our approach, we measured the overall performance penalty in terms of

additional processor cycles taken to complete the execution when using our security mechanisms. The overall performance penalty is depends on three factors: (i) extra memory accesses for fetching integrity information(signatures), (ii) encryption time for write accesses and decryption time for all cache misses resulting from data memory reads, and (iii) signature computation and validation time. The equation below depicts how the Cache miss penalty is computed based on the essential parameters: AES encryption and decryption (11 fpga cycles [20]) and signature computation (SHA1 taking 82 FPGA cycles [19] / 1 cycle for CRC computation).The cycle times correspond to the implementations of AES and SHA 1 on Xilinx Virtex II Pro.

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

We summarize the performance of our approach by showing the comparative performance penalty measured as the percentage increase in execution time when compared to the baseline configuration with no security mechanisms (i.e., plaintext execution). Figures 5 and 7 show the performance penalty for a 32-byte cache block size when using two different schemes for the integrity checking (*i.e.,* for computing signatures) – using secure hashing algorithm SHA-1 and a lightweight CRC hashing respectively. The memory access penalties come as an addition to the validation steps. As expected the compute intensive SHA-1 algorithm results in higher penalties while providing a larger degree of security. Due to the different data access patterns, our benchmarks exhibited varying amounts of performance penalties in both signature schemes. Using the compute intensive SHA-1 signature technique, resulted in a maximum increase of 26% in the execution time; this dropped to 4.5% when using the CRC scheme. Figure 6 shows the performance penalty for a 64-byte cache block size and using the SHA-1 algorithm. As expected the larger cache blocks incur less overhead (with a maximum of 16%), since the ratio of original data per signature is larger.

To provide complete protection against both code and data attacks on EED systems, we can add our code protection scheme presented in [15] (which resulted in performance penalties of at most 10%) to the data protection scheme presented in this paper. Thus, a complete system would result in the addition of the two performance penalties; *i.e.,* based on our experiments this led to performance penalties less than 30% using SHA-1 signatures and less than 15% using the simpler CRC method for signatures.

VII. Conclusions and Future Work

This paper proposed architectural support to protect application data from a class of EED attacks where the attacker has gained physical access to the system. Designed as a general mechanism for protecting both code and data our system achieves high security standards with relative modest performance penalties. The architecture and design of the secure hardware component is motivated by emerging trends in embedded systems: standard processor cores are augmented with FPGA fabric [18]. Our approach is transparent to software developers. The low performance overheads, ranging between 4.5% and 26% as shown in our experimental results through cycle accurate simulations, demonstrate the feasability of our approach. In the future, we hope to prototype our architecture on the Xilinx platform and thereby perform real-time experiments.

Acknowledgment

This research is partially supported by NSF grant ITR-025207 and AFOSR grant FA955006-1-0152.

References

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

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

[3] 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.

[4] O. Gelbart, P. Ott, B. Narahari, R. Simha, A. Choudhary, and J. Zambreno. CODESSEAL: A Compiler/FPGA Approach to Secure Applications, *IEEE Int. Conference on Intelligence and Security Informatics, 2005*.

[5] 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).

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

[7] 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).

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

[9] 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).

[10] 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.

[11] 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.

[12] Weidong Shi, Chenghuai Lu, Chenghuai Lu "Memory-centric Security Architecture" *High performance embedded architectures and compilers*, Barcelona, Spain, November 17-18, 2005.

[13] Emmett Witchel, Josh Cates, and Krste Asanovic "Mondrian Memory Protection" *ACM SIGARCH Computer Architecture News*, Volume 30 , Issue 5 (December 2002),Pages: 304 - 316.

[14] Shi, W. Fryman, J.B. Gu, G. Lee, H.-H.S. Zhang, Y. Yang, J. "InfoShield: a security architecture for protecting information usage in memory", The Twelfth International Symposium on High-Performance Computer Architecture,p. 222- 231, 2006

[15] E. Leontie, O. Gelbart, B. Narahari, R. Simha "Compiler-FPGA Technique to Detect Memory Spoofing in Encrypted-Execution Platforms, 6th Annual Security Conference, April 2007

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

[17] D. Kirovski, M. Drinic, M. Potkonjak: Enabling trusted software security. *Proc. ASPLOS 2002*.

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

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

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