CODESSEAL: Compiler/FPGA Approach to Secure Applications *

Olga Gelbart[1], Paul Ott[1], Bhagirath Narahari[1], Rahul Simha[1], Alok Choudhary[2] and Joseph Zambreno[2]

[1] The George Washington University, Washington, DC 20052 USA
[2] Northwesten University, Evanston, IL 60208 USA

Abstract. The science of security informatics has become a rapidly growing field involving different branches of computer science and information technologies. Software protection, particularly for security applications, has become an important area in computer security. This paper proposes a joint compiler/hardware infrastructure - CODESSEAL - for software protection for fully encrypted execution in which both program and data are in encrypted form in memory. The processor is supplemented with an FPGA-based secure hardware component that is capable of fast encryption and decryption, and performs code integrity verification, authentication, and provides protection of the execution control flow. This paper outlines the CODESSEAL approach, the architecture, and presents preliminary performance results.

1 Introduction

With the growing cost of hacker attacks and information loss, it is becoming increasingly important for computer systems to function reliably and securely. Because attackers are able to breach into systems in operation, it is becoming necessary not only to verify a program's integrity before execution starts, but also during runtime. Attackers exploit software vulnerabilities caused by programming errors, system or programming language flaws. Sophisticated attackers attempt to tamper directly with the hardware in order to alter execution. A number of software and software-hardware tools have been proposed to prevent or detect these kinds of attacks [1–3, 8]. Most of the tools focus on a specific area of software security, such as static code analysis or dynamic code checking. While they secure the system against specific types of attacks, current methods do not provide code integrity, authentication, and control flow protection methods that address attacks using injection of malicious code.

We propose a software/hardware tool - CODESSEAL - that combines static and dynamic verification methods with compiler techniques and a processor supplemented with a secure hardware component in the form of an FPGA (Field Programmable Gate Array) in order to provide a secure execution environment for fully encrypted execution. Figure 1 shows the architecture of our hardware.

* The research is supported in part by NSF grant CCR-0325207.

The main objective of the tool is to provide techniques to help proper authorization of users, to prevent code tampering and several types of replay, data, and structural attacks (such as control flow attacks), and to make it considerably harder for attackers to extract any information that could point to a potential vulnerability. CODESSEAL is also designed to support fully-encrypted executables to ensure confidentiality. Our preliminary experimental results reveal low performance overheads incurred by our software protection methods for many applications.

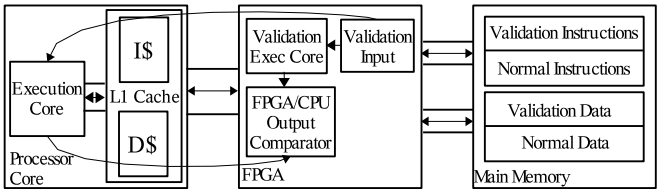

Fig. 1. Processor Core Supplemented with an FPGA

Several software techniques have been proposed for code security; these range from tamper resistant packaging,copyright notices, guards, code obfuscation [1–4] Software techniques typically focus on a specific type of vulnerability and are susceptible to code tampering and code injection by sophisticated attackers. Hardware techniques, including secure coprocessors and use of FPGAs as hardware accelerators [9, 10], are attractive because of the quality of service they provide but at the same time require substantial buy-in from hardware manufacturers and can considerably slow down the execution. Of greater relevance to our approach are the combined hardware-software approaches such as the XOM project [8], the Secure program execution framework (SPEF) [7], and hardware assisted control flow obfuscation [11]. In addition to other advantages over these techniques, such as the use of reconfigurable hardware in the form of FPGAs and working on the entire compiler tool chain, our approach addresses new problems arising from attacks on encrypted executables.

2 Our Approach: CODESSEAL

CODESSEAL (COmpiler DEvelopment Suite for SEcure AppLications) is a project focused on joint compiler/hardware techniques for fully encrypted execution, in which the program and data are always in encrypted form in memory. Encrypted executables do not prevent all forms of attack. Several types of replay, data and structural attacks, such as control flow attacks, are possible and can uncover program behavior. We term such attacks as EED attacks – attacks on *Encrypted Executables and Data*. EED attacks are based on exploiting structure

in encrypted instruction streams and data that can be uncovered by direct manipulation of hardware (such as address bus manipulation) in a well-equipped laboratory. To help detect such attacks, compilers will need to play a key role in extracting structural information for use by supporting hardware.

Fig. 2 describes the overall CODESSEAL framework. It has two main components – (1) static verification and (2) dynamic verification. At compile-time, static integrity and control flow information is embedded into the executable. The static verification module checks the overall integrity of the executable and also checks its signature. Upon success, the executable is launched and each block is dynamically verified in the FPGA for integrity and control flow. The CODESSEAL hardware architecture is shown in Figure 1. The architecture reconfigurability provided by an FPGA provides us with the ability to change the verification and cryptographic algorithms embedded in the hardware by simply reprogramming the FPGA. A detailed exposition of the static verification and authentication components of our framework, and experimental results, are provided in [6]. In this paper we focus on the dynamic verification component and present an implementation and preliminary performance study.

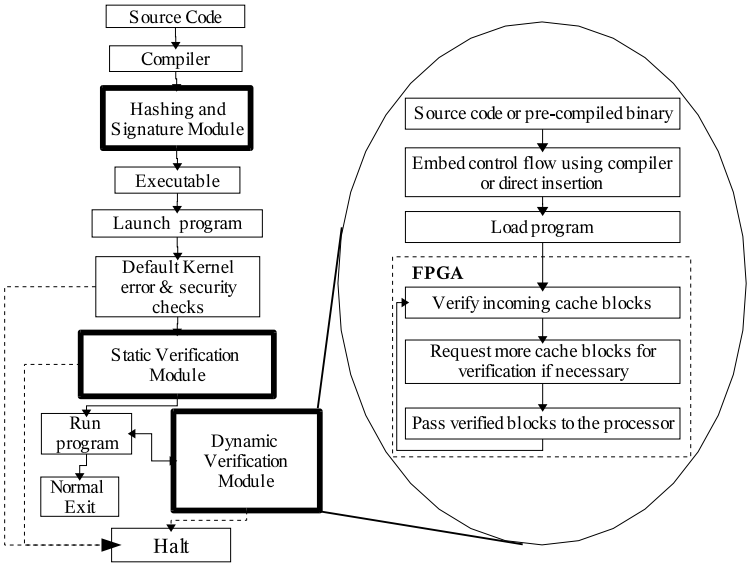

Fig. 2. CODESSEAL Framework

2.1 Dynamic Verification

If the static verification is performed successfully, the system allows launching of the executable. After this, the dynamic verification module is responsible for

preventing run-time attacks on the program. Our dynamic verification module has two components: (i) checking that code and data blocks have not been modified at run-time by an attacker and (ii) asserting the legal control flow in the program, i.e., any changes made to the control flow graph of the program is considered equivalent to code tampering and the program is halted.

Our technique involves the use of an FPGA placed between main memory and the cache that is closest to main memory (L1 or L2, depending on the system) – see Figure 2. The instructions and data are loaded into the FPGA in blocks and decrypted by keys that are exclusive to the FPGA. Thus, the decrypted code and data are visible "below" the FPGA, typically inside a chip, thereby preventing an attack that sniffs the address/data lines between processor and memory. The original code and data are encrypted by a compiler that uses the same keys. The assumption is that both FPGA loading and compilation occur at a safe site prior to system operation.

If full encryption is not used, instruction and data block hashes (using SHA-1, for example) can be maintained inside the FPGA and verified each time a new block is loaded. If the hash does not match the stored hash, the processor is halted. Even when full encryption is used, it is desirable to perform a hash check because a tampering attack can be used to disrupt execution without decryption. While this technique maintains code integrity it does not prevent structural (control flow) attacks which is the subject of our ongoing work.Our approach to preventing control-flow attacks embeds the control flow information, as captured by a control flow graph of the program, into the code. The signature of each block contains a hash of itself as well as control flow information denoting parent and child blocks. During the execution, only blocks whose hash and parent information is verified are permitted for execution. If a malicious block is introduced, its hash or control flow verification (performed in the secure FPGA component) will fail, thereby halting the program. Note that by using additional hardware to verify the program at runtime, we avoid adding additional code to the executable, thus preventing code analysis attacks.

Data tampering in encrypted systems is more complicated because a write operation necessitates a change in the encryption: data needs to be reencrypted on write-back to RAM. Also, because data can get significantly larger than code, a large set of keys might be needed to encrypt data, resulting in a key management problem.

2.2 Preliminary Experimental Results

Experimental Setup For the dynamic verification technique, we have currently implemented a scheme in which each instruction or data block of the executable is hashed using SHA1 algorithm. (Ongoing work explores the use of other cryptographic algorithms.) The hashes are stored in each block as well as in the FPGA. The FPGA performs hash verification as each block loads. Each block verification involves three steps: on L1 cache miss, a block is brought in, (a) its hash is calculated, (b) the "correct" hash is fetched from the FPGA memory and (c) the two hashes are compared.

The experimental setup was as follows. We used SimpleScalar version 3.0 processor simulator for the ARM processor and a gcc 3.3 cross-compiler. The ARM processor chosen to be represented by SimpleScalar had an ARM1020E core and ran at 300 MHz. The FPGA chosen was modeled after the Virtex-II XC2V800 and ran at 150 MHz and had at most 3 MB on onboard memory. The performance penalty for each of the three steps described above was: 6 processor cycles(3 FPGA cycles)for step (a) to calculate SHA-1 (maximum clock rate is 66 Mhz), 2 processor cycles(1 FPGA cycle) for step (b) (maximum clock rate for memory is 280 Mhz) and 2 processor cycles(1 FPGA cycle) for step (c) to compare hashes. The memory requirement on the FPGA was 22 bytes per cache block(20 per hash and 2 to map addresses to hashes).

Benchmark	Comment	No. instr	%Penalty (instr)	%Penalty (data)	%Penalty (both)
djpeg (MiBench)	136KB, 720x611	67.52M	.0389	2.2368	2.2757
	162KB,1265x2035	232.28M	.0117	4.3126	4.3242
rijndael(MiBench)	pdf 116.7KB	9.55M	.1238	.1100	.2337
	jpg 455KB	39.90M	.0297	.0263	.0560
susan(MiBench)	256KB, 512x512	71.16M	.0160	.7749	.7949
blowfish(MiBench)	pdf 116.7KB	20.21M	.0301	.0350	.0650
	jpg 455KB	83.86M	.0072	.0084	.0157
go(SPECINT2000)	6x6 board	26.22M	10.1072	7.7642	17.8724
	8x8 board	75.23M	9.5524	7.0244	16.5768
Transitive	16-64 vertices	15.24M	0.8	3.36	4.16
closure (DIS)	123-2048 edges				
	256-512 vertices	7.22B	0	42.89	42.89
	16384-196608 ed				
field (DIS)	Average for 5 runs	2.9B	0.02	0.08	0.08
pointer (DIS)	Average for 11 runs	1.09B	0.02	1.24	1.24

Table 1. Performance results for dynamic verification

Results and Analysis Dynamic verification was implemented for both code and data blocks and tested for a number of applications from the MiBench, DIS (Data intensive systems), and SPECINT2000 benchmarks. The control flow protection scheme is currently being implemented and thus not included in the results presented in this paper. Our experiments show an average of 3.97% performance penalty. The experimental results are summarized in Table 1, and reveal that our dynamic verification techniques result in very low performance degradation (overheads) in most cases. Some of the data intensive systems benchmarks, such as transitive closure, result in high overheads (of 42%) thereby motivating the need to study tradeoffs between security and performance.

3 Conclusions and Future Work

This paper proposed a tool - CODESSEAL - that combines compiler and FPGA techniques to provide a trusted computing environment while incurring low performance overhead in many benchmarks. The goal of our tool is to provide additional security to a computer system, where software authentication, integrity verification and control flow protection are particularly important. The tool is also designed to protected mission-critical applications against a hands-on attack from a resourceful adversary.

References

1. Cowan, C.: Software Security for Open-Source Systems. IEEE Security and Privacy (2003)
2. Chang, H., Attallah, M.J.: Protecting Software Code by Guards. Proceedings of the 1st International Workshop on Security and Privacy in Digital Rights Management (2000) 160-175
3. Colberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical Report. Dept of Computer Science, Univ. of Auckland (1997)
4. Fisher, M: Protecting binary executables. Embedded Systems Programming (2000) Vol.13(2).
5. Actel: Design security with Actel FPGAs. http://www.actel.com (2003)
6. Gelbart, O., Narahari, B., Simha, R.: SPEE: A Secure Program Execution environment tool using static and dynamic code verification. Proc. the 3rd Trusted Internet Workshop. International High Performance Computing Conference. Bangalore, India (2004)
7. Kirovski, D., Drinic, M., Potkonjak, M.: Enabling trusted software integrity. Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (2002) 108-120
8. Lie, D., Thekkath, C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., Horowitz, M.: Architectural support for copy and tamper resistant software. Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (2000) 168-177
9. Smith, S., Austel, V.: Thrusting trusted software: towards a formal model of programmable secure coprocessors. Proceedings of the 3rd USENIX Workshop on Electronic Commerce (1998) 83-98
10. Taylor, R., Goldstein, S.: A high-preformance flexible architecture for cryptography. Proceedings of the Workshop on Cryptographic Hardware and Software Systems (1999)
11. X. Zhuang, T. Zhang, S. Pande: Hardware assisted control flow obfuscation for embedded processors. Proc. of Int. Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES) 2004.