Performance Study of a Compiler/Hardware Approach to Embedded Systems Security *

Kripashankar Mohan, Bhagi Narahari, Rahul Simha and Paul Ott[1],Alok Choudhary and Joe Zambreno[2]

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

Abstract. Trusted software execution, prevention of code and data tampering, authentication, and providing a secure environment for software are some of the most important security challenges in the design of embedded systems. This short paper evaluates the performance of a hardware/software co-design methodology for embedded software protection. Secure software is created using a secure compiler that inserts hidden codes into the executable code which are then validated dynamically during execution by a reconfigurable hardware component constructed from Field Programmable Gate Array (FPGA) technology. While the overall approach has been described in other papers, this paper focuses on security-performance tradeoffs and the effect of using compiler optimizations in such an approach. Our results show that the approach provides software protection with modest performance penalty and hardware overhead.

1 Introduction

The primary goal of software protection is to reduce the risk from hackers who compromise software applications or the execution environment that runs applications. Our approach to this problem has been described in an earlier paper [1]. In this approach, bit sequences are inserted into the executable by the compiler that are then checked by supporting hardware during execution. The idea is that, if the code has been tampered, the sequence will be affected, thereby enabling the hardware to detect a modification. At this point, the hardware component can halt the processor.

Observe that the compiler can instrument executables with hidden codes in several ways. The most direct approach is to simply add codewords to the executable. However, this has the disadvantage that the resulting executable may not execute on processors without the supporting hardware, and may be easily detected by an attacker. Our approach is to employ the freedom that the compiler has in allocating registers. Because there are many choices in allocating registers, compilers can use these choices to represent binary codes that can then

* This work is supported in part by Grant CCR 0325207 from the National Science Foundation.

be extracted by the hardware. And because the register allocation is a valid one, the executable will run on processors that do not perform any checking.

The hardware support is needed so that the checking is itself not compromised, as is possible with software checking methods [10]. While it is generally expensive to build custom hardware support for this type of run-time checking, Field Programming Gate Array (FPGA) technology provides an attractive alternative. These programmable fabrics are today available with many commercial processors, and can easily be configured to perform the kind of checking during runtime. Because FPGA's are usually on-chip, and because they can be optimized to perform simple computations, they are also space and time efficient. Thus, the combination of compiler and FPGA technology makes the whole approach worthy of investigation. The purpose of this paper is to explore the performance of this approach.

Several factors can affect performance when actively checking an executable at runtime. The computation time in the FPGA depends on the lengths of the codes, their spread across the executable and how often the checking is performed. In particular, we consider the lengths of basic blocks and simple compiler techniques such as loop unrolling. We find that, overall, the approach imposes a modest penalty. At the same time, we find that loop-unrolling provides negligible performance benefit, thereby suggesting that other compiler techniques will be needed to further reduce the performance overhead.

Because a complete survey of software protection and security-driven compilation is presented in some of our earlier work [1, 9], we refer the reader to these papers for reviews of related work.

2 The Compiler/Hardware Approach

Figure 1 depicts the overall system architecture using our earlier approach [1]. The right side of Figure 1 shows a processor and FGPA on a single chip. As instructions stream into the chip, the FGPA secure component extracts the register information and uses the stream of registers to extract the hidden sequences, which are then checked. All the checking is performed by the FPGA itself.

To see how this works, consider a sample program as shown in Figure 2. The code on the left shows the output of standard compilation, before our register encoding is performed. The compiler approach replaces the standard register allocation algorithm with one that embeds keys. We use an even-numbered register to encode a 0 (zero) from the binary key, and an odd-numbered register to encode a 1 (one). This process continues until all the available "definition" registers of the basic block are assigned according to the key. The reverse process is performed in the FPGA.

In this paper, we consider two fundamental ways in which an embedded bit-sequence can be used. In the first method, the extracted bit sequence is compared against a pre-stored key in the FPGA. This, however, requires addressing the problem of key distribution. For systems in which such distribution or storage is not possible, a simpler (albeit less secure) approach can be used. In this case, the

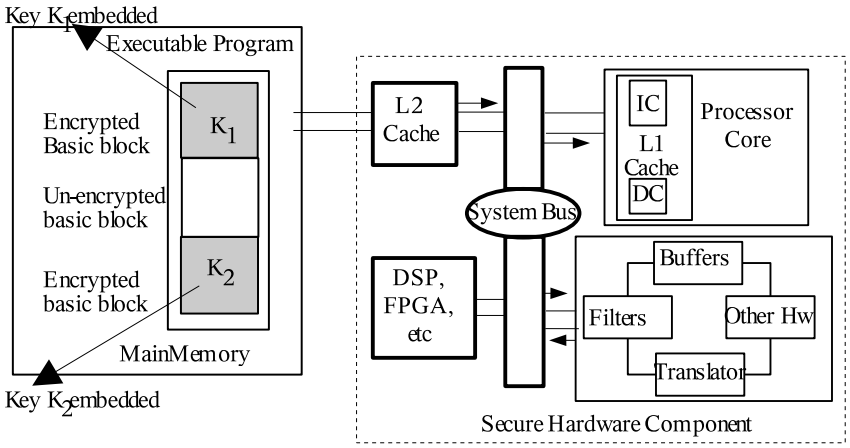

Fig. 1. System architecture

bit-sequence is compared against a specific (say, the first) opcode in the basic block. The overall technique relies on the fact that, when the code is tampered with, the bit sequence will be disrupted with high probability.

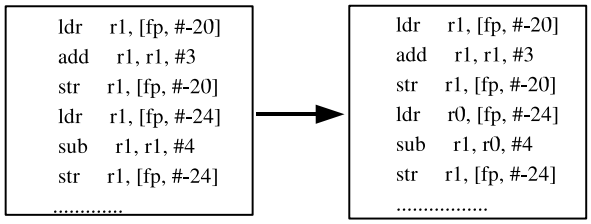

Fig. 2. Register encoding example with key 11010000011

Consider, for example, the second flavor mentioned above. In Figure 2, the opcode of the first instruction of a basic block yields a key of '11010000011000000000'. Since the first bit of the key is '1' the register assigned to the first instruction is 'R1'. From the use-def and def-use chain of the registers, the instructions that depend on the first instruction's register 'R1' are updated. In the second instruction, register 'R1' is once again chosen, as the bit value is '1' and the register is also redefined. An even register 'R0' is chosen for the fourth instruction from the available even registers, since the third bit of the key is '0'. This register's "use" locations are then updated. This process continues until all the registers

are modified based on the key chosen. A modified program with a new register sequence is thus obtained. For the above example, Figure 2 shows the result.

How is tampered code detected? Consider an insertion of instructions. Clearly, any instruction that uses registers is likely to disrupt the bit sequence with high probability, and will therefore be detected. Although not discussed here, it is easy to select opcodes that are at a fixed distance away, so that any insertion or deletion of instructions will result in a different opcode being checked, and therefore in failing the test. The security of a program can be increased by using private keys instead of opcodes of the instructions. This includes additional performance overhead of storing the keys in FPGA but can be tuned to meet the security objectives. Note that processors with caches will need to allow FPGA to probe the cache to fetch instructions of the encrypted basic blocks that span across multiple cache blocks.

3 Experimental Analysis

3.1 Experimental Framework

Our experimental framework uses the gcc cross-compiler targeting the ARM instruction set. The register allocation pass, the data flow analysis, loop unrolling and code generation pass of GCC were modified to incorporate the register-encoding scheme. The output of the compiler consists of the encrypted ARM executable and a description file for the FPGA.

The FPGA is simulated using the *Simplescalar* toolset [8]. The FPGA is placed between L1 cache and the main memory. It has access to the cache controller and can probe the cache. We assume that there is no L2 Cache. The FPGA is configured like a Virtex-II XC2V8000 component which is an ARM1020E type processor running at a chosen clock speed of 150 Mhz but with a processor clock rate of 300 MHz. The size of the on-board memory is 3MB with a memory access delay of 2 processor cycle (1 FPGA Cycle) and delay for comparing values is 2 processor cycles. The L1 cache is configured with 16 sets with the block size and associativity of cache being 32 and 64 respectively. The cache follows the Least Recently Used policy with a cache hit latency of one cycle. Main memory access latency is 15 and 2 cycles for the first and rest of the data respectively. The width of the FPGA operator is 8 bytes, nonsequential access latency being 10 cycles and sequential access latency being 10 cycles respectively. The FPGA validates each cache block that is being fetched from the main memory and placed into the L1 cache.

3.2 Benchmarks

Diverse benchmarks such as Susan (an image recognition package for Magnetic Resonance Images), Dijkstra (the well-known shortest path algorithm), Fast Fourier Transform, Rijndael (the AES encryption/decryption algorithm)and Patricia (a trie data structure) are used. We study the effect on overall system

performance using basic blocks of varying lengths for purposes of encoding the bit sequences. The results are normalized to the performance of unencrypted case as shown in Figure 3 (left side). The overall performance for most of the benchmarks is within 78% of the unencrypted case. Benchmarks Patricia and Dijkstra suffered a penalty of above 23%. This is because of instruction cache misses that occur in the FPGA that incur high cache miss penalty. Benchmarks Rijndael and FFT performed the best. Dijkstra suffered higher penalty due to large number of looping instructions being committed and it also runs longer.

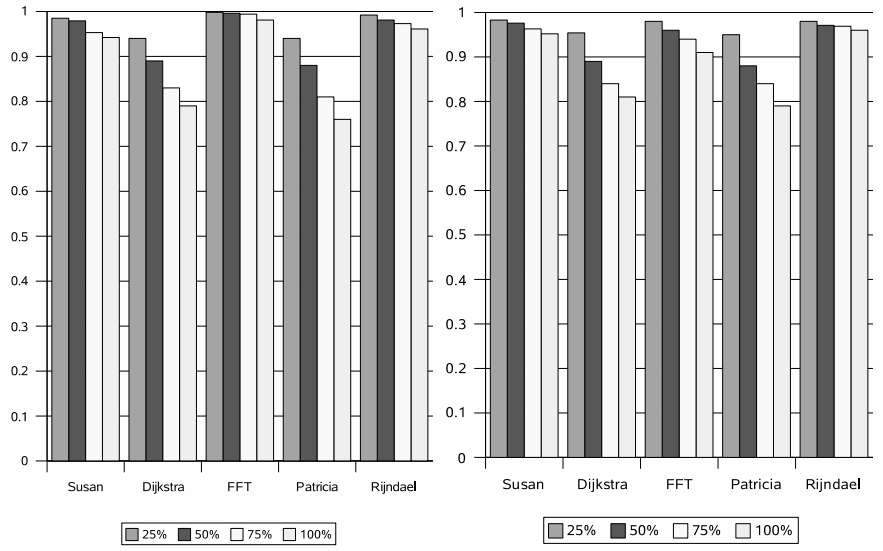

Fig. 3. Performance as a function of encryption of varying basic block lengths before(left side) and after(right side) performing loop unrolling on the benchmarks. (25%, 50%, 75% and 100% basic blocks are encrypted).

The effect of loop unrolling on the performance of the benchmarks was studied by selecting the basic blocks with longer instruction count. The main purpose of performing loop unrolling on the benchmarks is to increase the code size thereby increasing the number of instructions in the basic block, so that keys of greater length can be embedded into them. Further loop unrolling reduces loop overheads such as index variable maintenance and control hazards in pipelines, increases the number of statements in basic block to optimize and also improves the effectiveness of other optimizations such as common-subexpression elimination, software pipelining, etc. But its disadvantage is that the unrolled version of the code is larger than the rolled version, thereby having a negative impact on the performance on effectiveness of instruction cache. The effect of loop unrolling

on the benchmarks is shown in Figure 3 (right side). The optimization had little effect on Susan and Rijndael benchmarks. FFT suffered a 0.1% decrease in performance but performance of both computationally intensive Dijkstra and Patricia benchmarks is increased by nearly 3%. This shows that loop-unrolling provides negligible performance benefit, thereby suggesting that other compiler techniques will be needed to reduce the performance overhead. The security of the program is increased by loop unrolling, as the key length is increased since more number of registers are now available to embed keys due to the increase in code size.

4 Conclusions and Future work

Because embedded processors constitute an overwhelming share (above 90%) of the processor market, and because embedded devices are easily accessible to hackers, security has now become an important objective in the design of embedded systems. Pure-software approaches may not stop the determined hacker and pure-hardware approaches require expensive custom hardware. A combined compiler-FPGA approach offers the advantages of both at a fraction of the cost, while also offering backward-compatibility. The purpose of this paper was to study the performance of this approach using some well-known benchmarks, and to examine the effect of some compiler optimizations.

References

1. Zambreno, J., Choudhary, A., Simha, R., Narahari, B., Memon, A.: SAFE-OPS: A Compiler/ Architecture Approach to Embedded Software Security. ACM Transactions on Embedded Computing.
2. Chang, H., Atallah, M.: Protecting software code by guards. Proceedings of the ACM Workshop on Security and Privacy in Digital Rights Management. (2000) 160-175.
3. Collberg, C., Thomborson, C., and Low, D.: A taxonomy of obfuscating transformations. Dept of Computer Science, University of Auckland. Tech. Rep. 148 (1997).
4. Daeman, J. Rijmen, V.: The block cipher Rijndael. In Smart Card Research and Applications. Lecture Notes in Computer Science, vol. 1820. Springer-Verlag (2000) 288-296.
5. Dallas Semiconductor: Features, advantages, and benefits of button-based security. available at http://www.ibutton.com.
6. Gobioff, H., Smith, S., Tygar, D., Yee, B.: Smart cards in hostile environments. Proceedings of 2nd USENIX Workshop on Electronic Commerce. (1996) 23-28.
7. Necula, G.: Proof-carrying code. Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languagers. (1997) 106-119.
8. Todd Austin, Eric Larson, Dan Ernst.: SimpleScalar: An Infrastructure for Computer System Modeling. IEEE Computer. (February 2002).
9. Simha, R., Narahari, B., Choudhary, A., Zambreno, J.: An overview of security-driven compilation. Workshop on New Horizons in Compiler Analysis and Optimizations, Bangalore, India, (Dec 2004).
10. Wurster, G., Oorschot, P., Somayaji, A.: A generic attack on checksumming-based software tamper resistance. IEEE Symp. Security and Privacy, Oakland, CA (2005).