Flexible Software Protection Using Hardware/Software Codesign Techniques

Abstract

A strong level of trust in the software running on an embedded processor is a prerequisite for its widespread deployment in any high-risk system. The expanding field of software protection *attempts to address the key steps used by hackers in attacking a software system. In this paper we present an efficient and tunable approach to some problems in embedded software protection that utilizes a hardware/software codesign methodology. By coupling our protective compiler techniques with reconfigurable hardware support, we allow for a greater flexibility of placement on the security-performance spectrum than previously proposed mainly-hardware or software approaches. Results show that for most of our benchmarks, the average performance penalty of our approach is less than 20%, and that this number can be greatly improved upon with the proper utilization of compiler and architectural optimizations.*

1. Introduction

The emergence of embedded processors in high-risk devices has highlighted the need for strengthening their security. Indeed, even when considering consumer applications, the new-found ubiquity of embedded processors combined with the increased likelihood of networking capabilities being included in even the simplest of these devices has only intensified this focus on security. Hackers all over the world know that the key steps to attacking a software system is to first *understand* the software, and then to *tamper* with the software to enable a variety of full-blown attacks. The growing area of *software protection* aims to address the problems of code understanding and code tampering along with related problems of data tampering and authorization circumvention. As piracy is a substantial economic problem, software protection is considered one of the most significant outstanding challenges in security today [5]. The more generic threat addressed by this paper comes from hackers who modify program segments either statically or dynamically to allow the execution of an arbitrary amount of (potentially malicious) instructions while still maintaining much of the original intended program behavior. Given this backdrop, a necessary requirement for an embedded software protection scheme is that it makes it difficult for tampered code to execute, thereby providing a level of confidence that any code that is allowed to run is authorized by the user.

Our approach is motivated by the observation that prior approaches, themselves fairly recent, tend to lie at two extremes of the security-performance spectrum. At one end are highly secure hardware approaches using the Public-Key Infrastructure (PKI) that require a substantial buy-in from hardware manufacturers and can considerably slow-down execution speed. The other end relies on obscurity, by either mangling the code to make it less understandable or by burying checksum code-snippets in unlikely places. The latter approach is not as secure but does not degrade performance as much as the full PKI approach. These extremes invite an approach that allows system designers to position themselves where they choose on the security-performance spectrum.

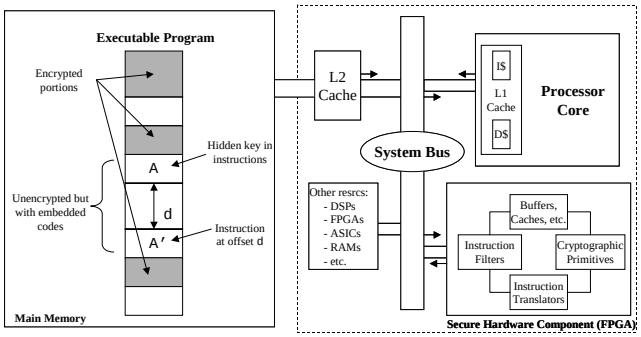

Figure 1. Conceptual view

Our proposed method works as follows (see Figure 1). The processor is supplemented with an FPGA-based *secure hardware component* that is capable of fast decryption and, more importantly, capable of recognizing and certifying strings of keys hidden in regular unencrypted instructions. To allow a desired level of performance, our compiler creates an executable with parts that are encrypted and parts that are unencrypted but are still tamper-proof. Thus, in Figure 1, the first part of the executable is encrypted and will be decrypted by the FPGA using a standard private key technique [6]. The second part of the executable shows an instruction block A containing a hidden key and, at a distance d from A, an instruction A'. Upon recognizing A, the FPGA will expect $A' = f(A)$ at distance d (where f is computed inside the FPGA); if the executable is tampered with, this match is highly unlikely to occur and the FPGA will halt the processor. The programmability of the FPGA together with the compiler's ability to extract program structure and transform intermediate code provides the broad range of parameters to explore security-performance tradeoffs.

We believe this approach has the following advantages: (1) the compiler's knowledge of program structure, it's execution profile, and the programmability of the FPGA allow for tuning of the security and performance of individual applications; (2) the approach is complementary to several software-based code checking techniques proposed recently [3, 4, 8]; (3) the use of FPGAs minimizes additional hardware design and is applicable to a large number of commercial processor platforms; (4) our processor/FPGA architecture is well-suited for future designs that utilize System-on-Chip (SoC) technology. Results demonstrate that our framework can be the successful basis for the development

of embedded applications that meet a wide range of security and performance requirements.

The remainder of this paper is organized as follows. In Section 2, we provide an overview of some of the more recent research in the field of software security. Section 3 goes into more detail about our codesign approach, explaining how a compiler can be modified to generate custom hardware and software descriptions that together can improve the overall system software security. In Section 4 we examine the performance implications of our approach, and we investigate the benefits of some hardware and software optimizations that can be applied within our experimental framework. Finally, in Section 5 we present our conclusions alongside a discussion of future optimizations and analysis techniques that are currently in development.

2. Related Work

Most other approaches to the general problem of software protection tend to focus on mainly-hardware or mainly-software solutions. *Secure coprocessors* are computational devices that can be trusted to execute their software in a trusted manner by running programs directly in an encrypted form. A number of secure coprocessing solutions have been designed and proposed, including systems such as IBM's Abyss [14] and μAbyss systems [13]. In [9] an architecture is proposed for tamper resistant software and a hardware implementation is provided, based on an execute-only memory (XOM) that allows instructions stored in memory to be executed but not manipulated. This type of approaches, while extremely secure, can greatly impact processor performance. Two key factors differentiate our work from the XOM approach. One distinction is that our architecture requires no changes to the processor itself. Also, our choice of reconfigurable hardware permits a wide range of optimizations that can shape the system security and resultant performance on a per-application basis. *Smart cards*, which can store sensitive computations and data with little or no direct user I/O, can also be viewed as a type of secure coprocessing [7, 12]. As noted in [3], a limitation of smart cards is that they can only be used to protect small fragments of code and data.

A common software-based protection technique is *obfuscation*, whereby code is deliberately mangled while maintaining correctness to make understanding difficult [4]. Obfuscating approaches range from simple encoding of constants to more complex methods that rearrange or transform code. In [3], the authors propose the concept of *guards*, pieces of executable code that typically perform checksums to protect against tampering. Similar dynamic self-checking techniques for improving tamper resistance are proposed in [8] and [1]. *Proof-Carrying Code* (PCC) is essentially a self-checking mechanism by which a host can verify code from an untrusted source [11]. Safety rules, as part of a theorem-proving technique, are used on the host as sufficient guarantees for proper program behavior. These techniques strongly rely on the security of the checksum computation itself; if these instructions are discovered by the attacker, they can be easily disabled. In many system architectures it is relatively easy to build an automated tool to reveal such software-based protective schemes. Our approach is complementary to many of these techniques, as the checking computations can themselves be strengthened by our HW/SW infrastructure.

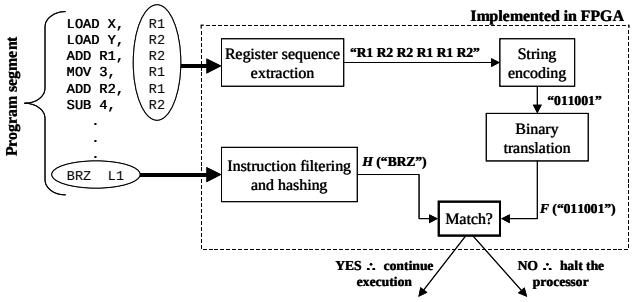

Figure 2. An illustrative example

3. Our Approach

3.1. An Illustrative Example

Consider a sample program as depicted on the left in Figure 2 and focus on the instruction stream. Initially, for illustration, we will not consider the complexity introduced by loops. The sequence of instructions comprising the instruction stream has instructions that use registers. We will extract one register in each register-based instruction. Then, the sequence of registers so used in the instruction stream is called the *register stream*.

In the example, part of the register stream shown is: R_1, R_2, R_2, R_1, R_1, R_2. The key observation is that this register stream is determined by the register allocation module of the compiler. In the FPGA hardware, the register stream is extracted from the instruction stream. In addition, the FPGA also extracts the opcode stream.

The example shows how R_1 encodes **0** and R_2 encodes **1** and therefore this particular sequence of registers corresponds to the code **0 1 1 0 0 1**. A binary translation component in the FPGA simply flips the bits (various transformations are possible) in the register code to result in the key: **1 0 0 1 1 0**. The key is then compared against a cryptographic function of the opcode stream. In the example above, an instruction filter module picks out an instruction following the register sequence (at distance d, as in Figure 1) and then compares (the function f in Figure 1) the key to the opcode. If a match occurs, the program segment is considered valid.

The example illustrates the main ideas:

- The compiler performs instruction filtering to decide which instructions in the opcode stream will be used for comparisons.
- The compiler uses the flexibility of register allocation to bury a code sequence in the register stream.
- The FPGA is programmed with an application-specific configuration which would include some knowledge of the overlying code structure.
- Upon execution, the instruction stream is piped through the secure FPGA component.

- The FPGA then extracts both the filtered opcodes and the register sequences for comparisons.
- If a violation is detected, the FPGA halts the processor.

If the sequence of instructions has been tampered with, there is a very high probability that the register sequence will be destroyed or that the opcode filtering will pick out a different instruction. For example, if the filtering mechanism picks out the sixth opcode following a register sequence, any insertion or deletion of opcodes would result in a failure.

3.2. General Approach

Our general approach is motivated by the observation that, since register allocation is done by the compiler, there is considerable freedom in selecting registers to allow for any code to be passed to the FPGA. The registers need not be used in contiguous instructions since it is only the sequence that matters. Note that other approaches could use instruction opcodes, immediate constants, or data addresses. However, because of its ease of use, and because of the independence of register allocation from the rest of compilation, using register sequences for our purpose makes the most intuitive sense. It is important to mention, that when examining a block of instructions to be protected using our scheme, it will often be the case that the compiler will lack a sufficient number of register-based instructions to perform the custom encoding. In this case, additional instructions which contain the desired sequence values but which otherwise do not affect processor state will need to be inserted by the compiler. This caveat comes with it a performance penalty which we examine in Section 4.1.

The register sequence can be used to encode several different items. For example, an authorization command can be inserted, after which other codes may be passed to the secure component. This technique can also be used to achieve obfuscation by using a secret register-to-register mapping in the FPGA. Thus, if the FPGA sees the sequence (R_1, R_2, R_1), this can be interpreted by the FPGA as an intention to actually use R_3. In this manner, the actual programmer intentions can be hidden through using a mapping customized to a particular processor. The complexity of the transformation can range from simple (no translation) to complex (private-key based translation). Such flexibility brings with it tradeoffs in hardware costs in terms of delay and performance.

It is important to again note that our approach can complement other techniques. Both code obfuscation and checksum-based protection schemes can be used alongside our approach. Clearly, the register allocation can be done independent of the program restructuring techniques typically used in obfuscation [4]. Similarly, our approach can be used with software checksum computations by ensuring that the checksum computation is itself secure, by preventing routing around the computation and interfering with the computation itself.

Finally, the strength of our approach is tunable in several ways. The key length and the mapping space can be increased, but at a computational cost. Furthermore, an optional secret key can be used to make f a cryptographic hash [2] for increased security. Similarly, by only using register codes that are examined by the FPGA, a lower level of security is provided, but the executable is left compatible with processors that do not contain the FPGA component. Most importantly, the computations performed in the FPGA are very efficient: there are counters, bit-registers and comparators. All of these operations can be performed within a few instruction cycles of a typical CPU.

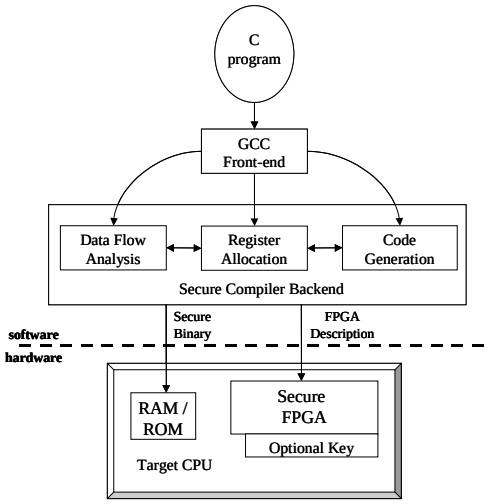

Figure 3. Compilation framework

4. Experimental Analysis

As previously mentioned, in the course of allocating registers to form key sequences, the compiler will often need to insert instructions into the executable. As the desired sequence length is increased this will consequently have a negative impact on performance. Also, for our scheme to work properly, the coarsest granularity of instructions that can encompass a single sequence is a program's basic block, defined as a sub-sequence of instructions that contains only one entry point and one exit point [10]. This is due to the fact that if register sequences were allowed to span basic blocks, the FPGA would have no guarantee that each instruction in a specific register sequence would be fetched exactly once per validation. Consequently, by varying both the register sequence length and the percentage of basic blocks that are encoded using our technique, we can evaluate the security/performance tradeoffs inherent in our approach.

4.1. Initial Evaluation

Figure 3 shows our current compilation framework. Using a modified version of the `gcc` compiler targeting the ARM instruction set, we implemented our register encoding schemes into the data-flow analysis, register allocation, and code generation phases of the `gcc` back-end. The output of our compiler is (1) – the encrypted application binary and (2) – a description file for the secure FPGA compo-

Benchmark	Source	Code Size	# Instrs
adpcm	MediaBench	8.0 KB	1.23M
g721	MediaBench	37.9 KB	8.67M
arm_fir	ARM AppsLib	44.0 KB	.301M
susan	MiBench	66.4 KB	2.22M
dijkstra	MiBench	42.5 KB	7.77M
fft	MiBench	69.2 KB	4.27M

Table 1. Selected benchmarks.

nent. We developed a custom hardware/software cosimulator in order to obtain performance results for our experiments. Our cycle-accurate simulator connects a commercial ARM core simulator to a commercial HDL simulator over a shared socket interface.

In order to test the effectiveness of our approach, we adapted a diverse set of benchmarks from a variety of embedded benchmark suites. These benchmarks are representative of the tasks that would be required of embedded processors used in multimedia and/or networking systems. More details can be found in Table 1.

For our initial experiments, we configured our simulator to model an ARM9TDMI core which contains sixteen, 32-bit general purpose registers. The ARM core is configured to operate at 200MHz and is combined with separate 8KB instruction and data caches. Setting the memory bus clock rate to be 66.7MHz, our cache miss latency before considering the FPGA access time is 150ns ($\sim$30 CPU cycles). This configuration is similar to that of the ARM920T processor. Our default FPGA model runs at a clock speed identical to that of the ARM core and requires an extra 3-5 cycles to process and decode an instruction memory access. Using our customized HW/SW cosimulator we first explored the effects of our approach on performance and resultant security by analyzing two main metrics: (1) the desired length of the register sequence; and (2) the selection criteria for inserting a sequence inside a suitable basic block.

Using our six benchmarks we simulated the effect of increasing the encoded register sequence length on the overall system performance when approximately 25% of the eligible basic blocks of each benchmark are secured using a random-selection algorithm. The results, as shown in Figure 4, are normalized to the performance of the unmodified case. As can be seen in the figure, these initial results demonstrate the potential security/performance tradeoffs inherent in our approach. Overall, for most of the benchmarks the performance is within 80% of the base case (no tamper proofing) when considering sequence lengths up to 16. However, when considering our most secure case (when the sequence length is 32), two of our benchmarks suffer a performance penalty of over 25%. This decreased performance is due to the fact that (1) the inserted instructions require extra cycles for their execution and that (2) the increased code size can lead to more instruction cache misses. This partially explains why our two largest benchmarks, *susan* and *fft*, performed the best of the set.

In Figure 5 we now consider the case where our register sequence length is kept at a constant value of 8, and the effect of the aggressiveness of our random basic block selection technique is examined. As we allow for a greater

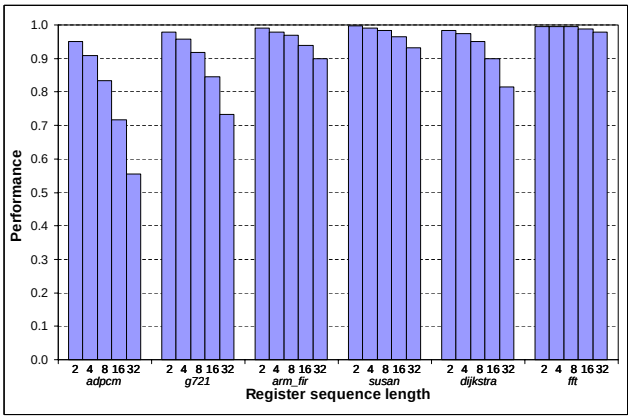

Figure 4. Performance as a function of the register sequence length, normalized to the performance of the unmodified benchmarks.

number of basic blocks to be encoded, we see that there is a limit to the performance of the resulting code. For the majority of our benchmarks, the performance in even the most secure case is within 75% of the base case. However it is to be noted that for our two smallest benchmarks (*adpcm* and *g721*), selecting more than 50% of the eligible basic blocks can have a drastic effect on performance. These results show that if it is possible for our compiler to select the right basic blocks to be encoded with an appropriate sequence length value, we will be able to keep our performance at an acceptable level while still increasing security.

4.2. Intelligent Basic Block Selection

In many cases, it would make sense for the application developer to select the basic blocks to apply the register code insertion technique on an individual basis. Appropriate targets for such a selection approach would be those that are most likely to be attacked; some examples are basic blocks that verify for serial numbers or that check to see if a piece software is running from its original CD. However, it may also be useful to cover a greater percentage of the entire executable with such a technique, as doing so would both increase the confidence that no block could be altered and would hinder the static analysis of the vital protected areas. To do this in a smarter fashion than the random selection approach of the previous section, it is necessary to first identify the root causes of the increase in total execution time.

Assuming a constant FPGA access latency, note that the key source of performance degradation are the additional instructions that may need to be added to form register sequences of the desired length. Given a basic block i of length l^i_{bblock} and a requested sequence length of l_{key} registers, the transformed basic block length can be written as:

$$l^{i'}_{bblock} = l^i_{bblock} + \alpha^i \cdot l_{key}\,, \qquad (1)$$

where $\alpha^i \in [0,1]$ is the percentage of requested sequence

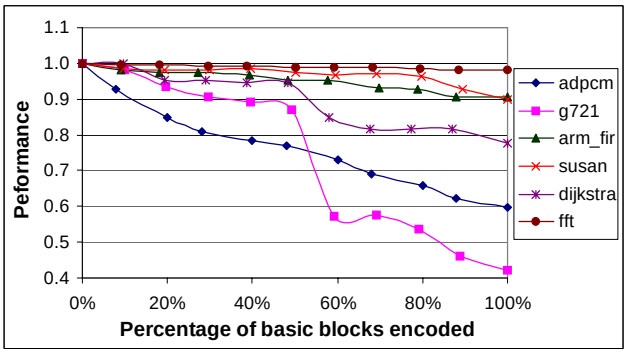

Figure 5. Performance as a function of the rate of encoding basic blocks, normalized to the performance of the unmodified benchmarks.

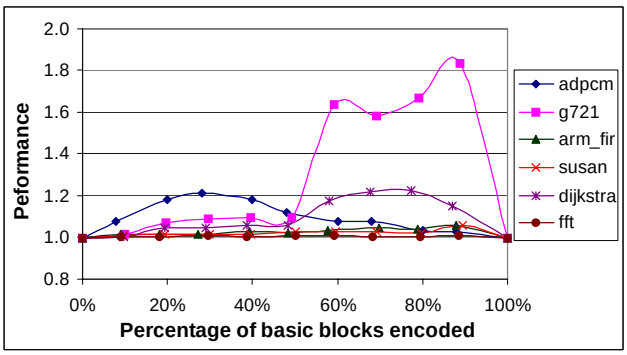

Figure 6. Performance improvements for iteration-based block selection optimization, relative to the randomly selected default.

length that cannot be encapsulated using register allocation. For example, for a register sequence length of 8, a basic block that can hide 6 of the needed key values in its original instructions would have an α^i value of 0.25, meaning that 2 additional instructions would need to be inserted to complete the sequence.

The number of extra pipeline cycles required by the execution of the inserted instructions is highly dependent on the loop structure of the original basic blocks. Consequently special consideration should be given to high-incidence blocks (i.e. blocks that are deeply nested in loops). Assuming that each basic block i is executed n^i_{iters} due to its containing loop structure, the total delay due to the execution of i can be estimated as:

$$t^i_{delay} = n^i_{iters} \cdot l^i_{bblock} \cdot CPI^i \,, \qquad (2)$$

where CPI^i is the average number of cycles per instruction required of the instructions in basic block i. If basic block i is selected for encoding using the register sequence technique, the new total execution time can be estimated as:

$$t^{i'}_{delay} = n^i_{iters} \cdot \left(l^i_{bblock} + \alpha^i \cdot l_{key}\right) \cdot CPI^{i'} \,. \qquad (3)$$

This equation shows that an obvious approach in selecting basic blocks for encoding is to sort by increasing number of iterations. In practice, however, there will be several blocks in an application that will not be fetched at all during normal program execution. Applying dead code elimination will not necessarily remove these blocks, as they are quite often used for error handling or other functionality that is rarely needed but still vital. For this reason the blocks where $n^i_{iters} = 0$ are placed at the end of the ordered list of eligible basic blocks. Also, it will be very likely that ties will exist between different basic blocks with the same n^i_{iters} value. A suitable heuristic to sort these sets of blocks can be developed by considering that a basic block with a larger l^i_{bblock} value will have a relatively smaller number of additional instruction cache misses after a register sequence encoding when compared with other blocks with the same n^i_{iters} value but with a shorter original block length. For this reason blocks can be effectively sorted first by non-zero increasing n^i_{iters} values and then by decreasing l^i_{bblock} values.

To estimate the number of iterations for the basic blocks in the selected benchmarks, profiling runs were performed that fed back the individual loop counts to the basic block selection module of the compiler. As this approach can potentially lead to a significant increase in compile time, the profiling experiments were ran using smaller input sets for the benchmarks, with the goal of not letting profiling information increase the compiler run-time more than 6 times. This increase is acceptable. Embedded systems can tolerate much larger compilation times than their general-purpose counterparts since the resulting programs are used so many times without further compilation.

Figure 6 shows the general trends seen when the tests of the previous section are re-run with the iteration-based block selection policy. These results show that by initially selecting blocks with low iteration counts, the large performance degradations can be delayed until only the highest level of security is required. As can be seen from the figure, the performance gains due to intelligent basic block selection are as large as 80% in one case (*g721*), averages between 5-20% for the majority of the benchmarks, and is negligible for the *fft* benchmark. The reason the *g721* benchmark performs so well is that the code size is relatively small, and that an overwhelming percentage of the total run-time is concentrated in a just a few basic blocks.

4.3. FPGA-based Instruction Caching

Given the reconfigurable nature of FPGAs, it is interesting to explore architectural optimizations that could be implemented to improve performance, while still maintaining a desirable level of security. In this section we investigate using the FPGA as a program-specific secondary level of instruction cache. As an example, consider a configuration where the FPGA caches only the instructions that are fetched inside selected basic blocks. After the entire block is validated, this smaller cache can then return the instruction values for future requests without requiring an expensive access to main memory or any other cycles spent in decoding or decryption. This technique would gain in

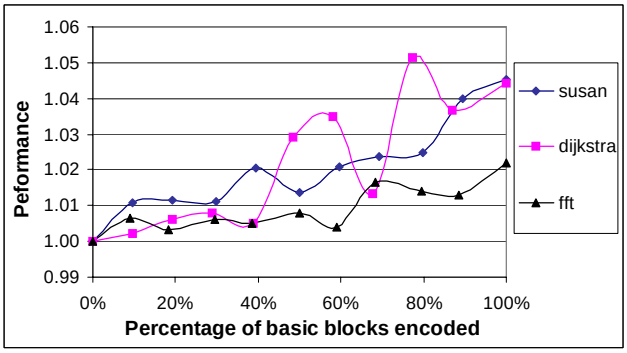

Figure 7. Performance improvements for the basic block caching optimization.

preference under a configuration where the FPGA tamper-checking algorithm requires a relatively large number of computational cycles. An important feature of this approach is that the compiler generates application-specific FPGA configurations that include basic block information.

In evaluating this architectural optimization we examined the effect on performance for three benchmarks when the sequence length was kept constant at 8 and the percentage of basic blocks to be selected (randomly) was increased from 0-100%. Figure 7 shows a summary of the instruction memory hierarchy performance of this technique, normalized to the case where no caching is performed on the FPGA. As can be seen from the figure, the basic block caching approach was limited to a 5% speedup. It is interesting to note however that this approach does demonstrate near-linear speedups as the aggressiveness of the encoding algorithm is increased. This is an important result, as it implies that if we tune other system parameters to allow for even more cryptographic strength, this type of approach will begin to perform favorably well when compared to a general caching strategy.

5. Conclusions and Future Work

The rapidly increasing use of embedded processors in critical applications motivates the need for an intensified scrutiny of the security of the system software. In this paper we presented a novel approach to embedded software protection that utilizes a hardware/software codesign methodology. The main benefit of our approach is the flexibility it allows the embedded application designer in terms of positioning on the security-performance spectrum. Our results show that we are able to improve application security at a nominal cost to performance.

Note that for our approach each protected program requires a unique FPGA representation needing permanent storage, and that this file would potentially need to be reloaded for each processor task switch. As reverse engineering and selectively reprogramming an FPGA is a considerably difficult undertaking when compared to an equivalent attack on software, we can state with some confidence that the choice of FPGA hardware increases application security when compared to the previously discussed mainly-software approaches.

Several improvements to our framework are being developed. In order to more accurately measure the effect of different FPGA configurations on clock rates and performance, we are considering adding a synthesis path to our compiler. Also, as our current approach operates only on source code, our hope is in the future to utilize decompilation techniques to allow our software protection methodology to be applicable to pre-compiled code and libraries.

References

[1] D. Aucsmith. Tamper-resistant software: An implementation. In *Proceedings of the 1st International Workshop on Information Hiding*, pages 317–333, May 1996.

[2] M. Bellare, R. Canetti, and H. Krawczyk. Message authentication using hash functions: the HMAC construction. *RSA Laboratories' CryptoBytes*, 2(1), Spring 1996.

[3] H. Chang and M. Atallah. Protecting software code by guards. In *Proceedings of the ACM Workshop on Security and Privacy in Digital Rights Management*, pages 160–175, Nov. 2000.

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

[5] Computer Security Institute and Federal Bureau of Investigation. CSI/FBI 2002 computer crime and security survey. available at http://www.gocsi.com, Apr. 2002.

[6] J. Daeman and V. Rijmen. The block cipher Rijndael. In J.-J. Quisquater and B. Schneier, editors, *Smart Card Research and Applications*, volume 1820 of *Lecture Notes in Computer Science*, pages 288–296. Springer-Verlag, 2000.

[7] H. Gobioff, S. Smith, D. Tygar, and B. Yee. Smart cards in hostile environments. In *Proceedings of the 2nd USENIX Workshop on Electronic Commerce*, pages 23–28, Nov. 1996.

[8] B. Horne, L. Matheson, C. Sheehan, and R. Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *ACM Workshop on Security and Privacy in Digital Rights Management*, pages 141–159, Nov. 2001.

[9] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 168–177, Nov. 2000.

[10] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[11] G. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languagers*, pages 106–119, Jan. 1997.

[12] B. Schneier and A. Shostack. Breaking up is hard to do: modeling security threats for smart cards. In *Proceedings of the USENIX Workshop on Smartcard Technology*, pages 175–185, May 1999.

[13] S. Weingart. Physical security for the mABYSS system. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 52–58, Apr. 1987.

[14] S. White and L. Comerford. ABYSS: A trusted architecture fo software protection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 38–51, Apr. 1987.