# SAFE-OPS: An Approach to Embedded Software Security

JOSEPH ZAMBRENO and ALOK CHOUDHARY
Northwestern University
RAHUL SIMHA and BHAGI NARAHARI
The George Washington University
and
NASIR MEMON
Polytechnic University

The new-found ubiquity of embedded processors in consumer and industrial applications brings with it an intensified focus on security, as a strong level of trust in the system software is crucial to their widespread deployment. The growing area of *software protection* attempts to address the key steps used by hackers in attacking a software system. In this paper, we introduce a unique approach to embedded software protection that utilizes a hardware/software codesign methodology. Results demonstrate that this framework can be the successful basis for the development of embedded applications that meet a wide range of security and performance requirements.

Categories and Subject Descriptors: C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Adaptable architectures*; C.3 [**Special-Purpose and Application-Based Systems**]: Real-Time and Embedded Systems

General Terms: Design, Measurement, Performance, Security

Additional Key Words and Phrases: Software protection, HW/SW codesign

## 1. INTRODUCTION AND MOTIVATION

Software protection is one of the most significant outstanding problems in security today [Computer Security Institute and Federal Bureau of Investigation

This work was supported in part by the National Science Foundation (NSF) under grant CCR-0325207 and also by an NSF graduate research fellowship.
Authors' addresses: J. Zambreno and A. Choudhary, Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208; email: {zambro1,choudhar}@ece.northwestern.edu; R. Simha and B. Narahari, Department of Computer Science, The George Washington University, Washington, DC 20052; email: {simha,narahari}@gwu.edu; N. Memon, Department of Computer and Information Science, Polytechnic University, Brooklyn, NY 11201; email: memon@poly.edu.

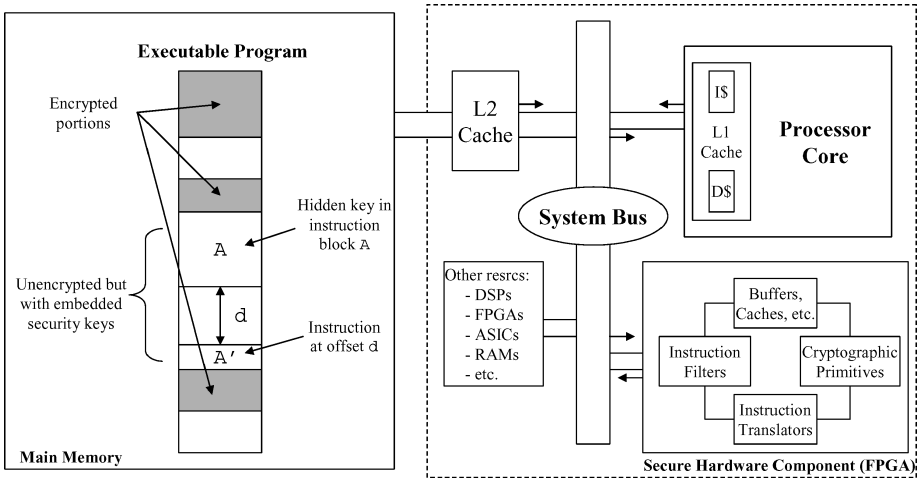


Fig. 1. Conceptual view.

2002]. Because most successful attacks involve tampering with executable instructions, the related problems of *code understanding*, *data tampering*, and *authorization circumvention* together with *code tampering* form the four main types of attacks on software systems. In addition, while software protection is an important issue for desktop and servers, the fact that over 97% of all processors are embedded processors [Hoffmann et al. 2002] shows the importance of protecting software on embedded systems. Imagine an attack on a key networked microcontroller used in transportation systems: the resulting tampered executables can cause large-scale disruption of mechanical systems. Such an attack can be easily replicated because embedded processors are so numerous. Efforts in the area of software protection tend to lie at one of two extremes: a "camouflage" approach in which software is obfuscated or in which software checks are periodically embedded, or a "Fort Knox" approach in which the processor is architected with sophisticated public-key cryptographic hardware and executables are encrypted. As is pointed out later, the "camouflage" approach may only slow down hackers whereas the "Fort Knox" approach requires a new processor technology.

This paper presents an approach whose level of security lies between the two extremes and can be tuned within this range. Furthermore, by exploiting the wide availability and reprogrammability of FPGA (field-programmable gate array) hardware, these techniques will be applicable in the near future (3–5 years) and are compatible with system-on-chip (SoC) designs that feature processor cores, interconnects, and programmable logic. The 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 (execution speed), the compiler creates an executable with parts that are encrypted (the "slow" parts) 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 (secret) key technique [Daeman and Rijmen 2000]. The second part of the executable shows 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 key sequences can be hidden within both instructions and data. The association of $A'$ with $A$ is hidden even from invasive electronic probing. The FPGA waits for the processor to fetch $A$ and $A'$ during program execution before checking the key. The programmability of the FPGA together with the compiler's ability to extract program structure and transform intermediate instructions provide the broad range of parameters to enable security-performance trade-offs.

This approach has the following advantages: (1) it simultaneously addresses the four main types of attacks on software integrity; (2) the compiler's knowledge of program structure, its execution profile, and the programmability of the FPGA allow for tuning the security and performance for individual applications; (3) the approach is complementary to several software-based instruction validation techniques proposed recently [Chang and Atallah 2000; Collberg et al. 1997; Horne et al. 2001]; (4) the hardware required in this scheme is simple and fast; (5) the use of FPGAs minimizes additional hardware design and is applicable to a large number of commercial processor platforms; (6) this processor/FPGA architecture is well suited for future designs that utilize SoC technology. Experiments show that for most of the selected benchmarks, the average performance penalty for this approach is less than 20%, and that this number can be greatly improved upon with the proper utilization of compiler and hardware optimizations.

The remainder of this paper is organized as follows. In Section 2, background is provided into the problem of software security, with an overview of the more recent research in that field. Section 3 provides more details about the SAFE-OPS approach, explaining how the compiler and FPGA interact in a tunable fashion to improve system software security. In this section, experimental results are also presented detailing the inherent performance/security trade-offs by examining several embedded application benchmarks. Section 4 investigates the performance benefits of some hardware and software optimizations that can be applied within the experimental framework. Finally, in Section 5 conclusions are presented, with a discussion of future optimizations and analysis techniques that are currently in development.

## 2. RELATED WORK

This section reviews past work in the general area of software security. As will be shown, most approaches tend to focus on purely hardware or purely software solutions.

### 2.1 Mostly-Hardware Approaches

*Tamper-resistant packaging* can coat circuit boards or encase the entire device, such as the iButton developed by Dallas Semiconductor [1999]. Alternatively, a

*custom processor* can be used with special-purpose hardware that makes it difficult to decompile code, and extremely difficult to insert tampered code. However, this approach is much more expensive than providing tamper-resistant packaging.

*Secure coprocessors* are computational devices that can be trusted to execute their software in a trusted manner. Programs, or parts of the program, can be run (in an encrypted form) on these devices thus never revealing the code in the untrusted memory and thereby providing a tamper resistant execution environment for that portion of the code. A number of secure coprocessing solutions have been designed and proposed, including systems such as IBM's Citadel [White et al. 1991], Dyad [Tygar and Yee 1993; Yee 1994; Yee and Tygar 1995], the Abyss and $\mu$Abyss systems [Weingart 1987; Weingart et al. 1990; White and Comerford 1987], and the commercially available IBM 4758 which meets the FIPS 140-1 level 4 validation [IBM 2002; Smith 1996; Smith and Weingart 1999]. Distributed secure coprocessing is achieved by distributing a number of secure coprocessors and some have augmented the Kerberos system by integrating secure coprocessing into it [Itoi 2000]. In Lie et al. [2000], 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. The machine works on encrypted instructions and the XOM machine decodes the session key, decodes instructions from external memory using the session key, and provides partitioned storage for the XOM code. For x86 Intel processors, they estimate that a memory-bound computation will have a less than 50% slowdown. Finally, several companies have formed the so-called Trusted Computing Platform Alliance [2003] to provide hardware–software solutions for software protection.

*Smart Cards*. Smart cards can be viewed as a type of secure coprocessing; a number of studies have analyzed the use of smart cards for secure applications [Gobioff et al. 1996; Kommerling and Kuhn 1999; Neve et al. 1999; Schneier and Shostack 1999; Smith 1996]. Sensitive data and critical instructions can be stored on smart cards but they offer no direct I/O to the user. Most smart card applications focus on the secure storage of data although studies have been conducted on using smart cards to secure an operating system [Clark and Hoffman 1994]. As noted in Chang and Atallah [2000], smart cards can protect only small fragments of code and data. In addition, as noted in Kocher [1995] and Smith [1996], low-end devices derive their clocks from the host and thus are susceptible to attacks that analyze data-dependent timing.

### 2.2 Mostly Software Approaches

A survey of several software protection techniques appears in Fisher [2000]. A survey of the broader area of software security in the context of digital rights management (DRM) appears in Chang et al. [2001] and Wyant [2001]. The SAFE-OPS approach of embedding codes resembles steganography [Johnson and Katzenbeisser 1999] in some ways but that the goals and details are quite different.

*Copyright Notice and Watermarking*. The oldest "prevention" technique, embedding a copyright into the code, does not prevent a hacker from actually using the code or modifying it. A survey and taxonomy of watermarking techniques is given in Collberg and Thomborson [1999, 2002].

*Obfuscation*. In this technique, code is deliberately mangled while maintaining correctness to make understanding difficult—a survey of obfuscation techniques appears in Collberg et al. [1997]. Obfuscation techniques range from simple encoding of constants to more complex methods that re-arrange or transform code [Collberg et al. 1997, 1998]. Other authors [Wang et al. 2000, 2001] also propose transformations to the code that make it difficult to determine the control flow graph of the program, and show that determining the control flow graph of the transformed code is NP-hard. Theoretical limitations are discussed in Barak et al. [2001]. However, many obfuscation techniques can be attacked by designing tools that automatically look for obfuscations. Another approach to attacking obfuscation techniques is to emulate the code in a debugger and to identify vulnerabilities in the code step-by-step using the debugger. The SAFE-OPS approach is complementary to obfuscation techniques.

*Code-Checksums*. In Chang and Atallah [2000], the authors propose the concept of *guards*, pieces of executable code that typically perform checksums to guard against tampering. In Horne et al. [2001], the authors propose a dynamic self-checking technique to improve tamper resistance. The technique consists of a collection of "testers" that test for changes in the executable code as it is running and report modifications. A tester computes a hash of a contiguous section of the code region and compares the computed hash value to the correct value. An incorrect value triggers the response mechanism. They note that performance is invariant until the code size being tested exceeds the size of the L2 cache. A marked deterioration in performance was observed after this occurred. In Aucsmith [1996] a self-checking technique is presented in which embedded code segments verify the integrity of the program during runtime. These techniques strongly rely on the security of the checksum computation itself. If these checksum computations are discovered by the attacker, they are easily disabled. Moreover, in many system architectures, it is relatively easy to build an automated tool to reveal such checksum-computations. For example, a control-flow graph separates instructions from data even when data is interspersed with instructions; then, checksum computations can be identified by finding code that operates on code (using instructions as data). This problem is acknowledged but not addressed in Horne et al. [2001].

*Proof-Carrying Code*. Proof-carrying code (PCC) is a technique by which a host can verify code from an untrusted source [Appel and Felten 1999; Baifanz et al. 2000; Bauer et al. 2001; Necula 1997, 2003; Necula and Lee 1996]. Safety rules, as part of a theorem-proving technique, are used on the host as sufficient guarantees for proper program behavior. Applications include browser code (applets) [Baifanz et al. 2000] and even operating systems [Necula and Lee 1996]. One advantage of proof-carrying software is that the programs are self-certifying, independent of encryption or obscurity. The PCC method is

essentially a self-checking mechanism and is vulnerable to the same problems that arise with the code checksum methods discussed earlier; in addition they are static methods and do not address changes to the code after instantiation.

*Custom OS*. Another software technique is to create a system of customizable operating systems wherein much of the security, checksumming, and obfuscation are hidden. However, in practice this solution will take longer to find acceptance for several reasons. First, it is expensive to maintain multiple flavors of operating systems. Second, the main vendor of entrenched operating systems fiercely protects code sources—a political act may be needed to require such customizability. Third, the installed base of operating systems is already quite high, which would leave many existing systems still open to attack.

### 2.3 FPGA-Based Approaches

FPGAs have been used for security-related purposes in the past as hardware accelerators for cryptographic algorithms. Along these lines Dandalis and Prasanna [Dandalis et al. 2000; Prasanna and Dandalis 2000] have led the way in developing FPGA-based architectures for internet security protocols. Several similar ideas have been proposed in Taylor and Goldstein [1999] and Kaps and Paar [1998]. The FPGA manufacturer Actel [2003a] offers commercial IP cores for implementations of the DES, 3DES, and AES cryptographic algorithms. These implementations utilize FPGAs not only for their computational speed but for their programmability; in security applications the ability to modify algorithmic functionality in the field is crucial.

In order for an FPGA to effectively secure an embedded or any other type of system, there is a requisite level of confidence needed in the physical security of the FPGA chip itself. In an attempt to raise consumer confidence in FPGA security, Actel [2003b] is currently developing new antifuse technologies that would make FPGAs more difficult to reverse-engineer. This added physical security greatly enhances the value of the SAFE-OPS approach, since as hardware technologies improve to make the "secure component" more secure, the level of trust of the software running on a target embedded system significantly increases.

The approach described in this paper is unique in its utilization a combined hardware and software technique, and that it allows designers tremendous flexibility in terms of their positioning in the security-performance spectrum. For these reasons this work, while certainly applicable to a variety of platforms including desktops and server platforms, is well suited for embedded systems. Ultimately, the proposed research framework, independent of the particular software protection details, will provide practical tools to manage the security-performance trade-off; other protection strategies can be tested as plug-ins into these tools.

## 3. THE SAFE-OPS APPROACH

Our approach, called SAFE-OPS (software/architecture framework for the efficient operation of protected software), will consider the following broad themes. First, designers would like the ability to fine-tune the level of security for a

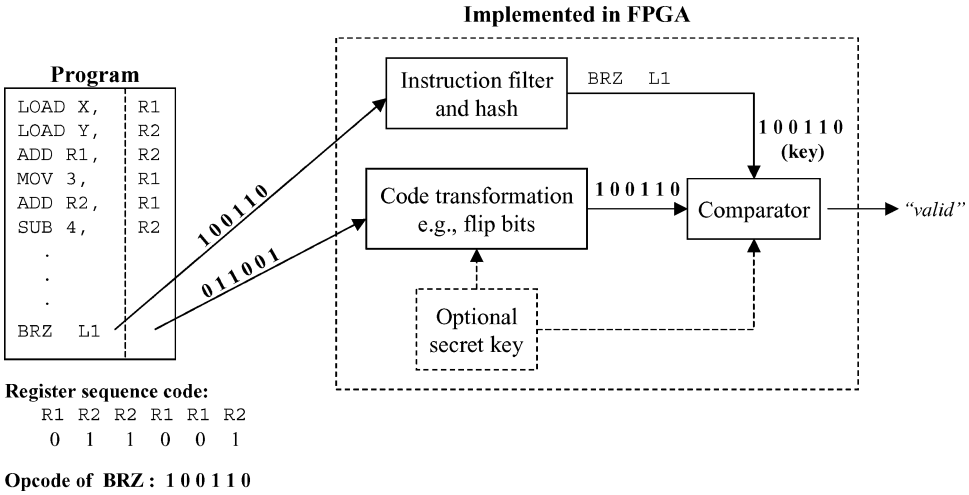


Fig. 2. A register stream example.

desired level of performance. Second, it may not be practical to store a secret key within the FPGA in small embedded processors; for such systems, a simpler approach is recommended. Third, for performance reasons, it is possible to inhibit tampering without necessarily addressing code understanding.

### 3.1 A Register Stream Example

Consider the sample program depicted on the left in Figure 2 and focus on the instruction stream. Initially, for illustration, the complexity introduced by loops is not considered. The instruction stream contains instructions that use registers. Isolating one register in each register-based instruction will extract a sequence from the instruction stream that can be called the *register stream*.

In the example of Figure 2, the 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.

In the example, $R_1$ encodes "0" and $R_2$ encodes "1" and therefore the sequence of registers $(R_1, R_2, R_2, R_1, R_1, R_2)$ corresponds to the code **0 1 1 0 0 1**. As an illustration, a code-transformation component in the FPGA simply flips the bits (other transformations are possible) in the register stream 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 register-stream key to the opcode. If a match occurs, the code is considered valid.

The example illustrates the key 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 key sequence in the register stream.
—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 code has been tampered with, there is a very high probability that the register sequence will be destroyed or that the opcode filtering will select a different instruction. For example, if the filtering mechanism picks the sixth opcode following a register sequence, most insertions or deletions of opcodes will result in a failure.

The example above shows how compiler-driven register keys can be used for efficiently ensuring code integrity, authorization, and obfuscation. By including these keys in compiler-driven data allocation, the fourth goal of data integrity can also be included.

### 3.2 General Approach

The 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 key to be passed to the FPGA. The registers need not be used in contiguous instructions since it is only the sequence that matters. 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 this purpose makes the most intuitive sense. When examining a block of code to be encrypted using this scheme, often the compiler will lack a sufficient number of register-based instructions to encode the desired custom key. In this case, "place-holder" instructions which contain the desired sequence values but which otherwise do not affect processor state, can be inserted by the compiler. Figure 3 shows a sample basic block where the desired register sequence values are obtained using a combination of register allocation and instruction insertion. Introducing instructions adds a performance penalty that is examined in Section 3.3.3.

The register-sequence can be used to encode several different items. For example, an authorization code can be encoded, after which other codes may be passed to the secure-component. This technique can also be used to achieve code 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 in terms of speed and cost.

```
0x00d0: CMP    r0, r12
0x00d4: SUBLT    r7,r1,r0
0x00d8: LDRLT    r7,[r14,r7,LSL #2]
0x00dc: LDRLT    r2,[r13,r1,LSL #2]
0x00e0: LDRLT    r3,[r4,r0,LSL #2]
0x00e4: ADDLT    r0,r0,#1
0x00e8: MLALT    r2,r7,r3,r2
0x00ec: STRLT    r2,[r13,r1,LSL #2]
0x00f0: BLT    0xd0                                  Original basic block
```

Desired sequence: **1 0 0 1 0 1 1 0**
Register encoding: r10 = 0, r11 = 1

```
0x00d0: CMP    r0, r12
0x00d4: ADD    r11, r12, r13    // Inserted instruction will not change CPU state
0x00d8: SUBLT    r10,r1,r0    //  Register change r7 → r10
0x00dc: LDRLT    r10,[r14,r10,LSL #2]
0x00e0: LDRLT    r11,[r13,r1,LSL #2]    // Register change r2 → r11
0x00e4: LDRLT    r3,[r4,r0,LSL #2]
0x00e8: ADDLT    r0,r0,#1
0x00ec: ADDLT    r10, r10, #0    // Careful insertion of harmless instruction
0x00f0: MLALT    r11,r10,r3,r11
0x00f4: STRLT    r11,[r13,r1,LSL #2]
0x00f8: SUBLT    r10, r10 #0    // Additional inserted instruction
0x00fc: BLT    0xd0                                  Modified basic block
```

Fig. 3. Obtaining a desired register sequence value. In this example the sequence is obtained by reallocating registers in the instructions that use the registers r7 and r2 to those that use r10 and r11.

The SAFE-OPS approach can complement other approaches. Both code-obfuscation and checksum-based code-tampering approaches can be used simultaneously. The register allocation can be done independent of the code restructuring techniques typically used in code-obfuscation [Collberg et al. 1997]. As an example, software checksum computations can be strengthened by ensuring that the checksum computation is itself secure. The start address of the checksum computation can be passed into the secure-component using a register-sequence code. The FPGA will expect the checksum computation to be executed at that starting address. Next, by inspecting successive addresses during the checksum computation, the secure-component will ensure that a predetermined range of addresses are used in the checksum computation. This approach prevents two attacks on the checksum computation: (1) routing around the computation and (2) interfering with the checksum computation.

The strength of this approach is tunable in several ways. The key length and the mapping space can be increased, but at a computational cost. An optional secret key can be used to make $f$ a cryptographic hash [Bellare et al. 1996] for increased security. By only using register codes that are examined by the FPGA, a lower level of security is provided, but the executable instructions are left compatible with processors that do not contain the FPGA component. 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

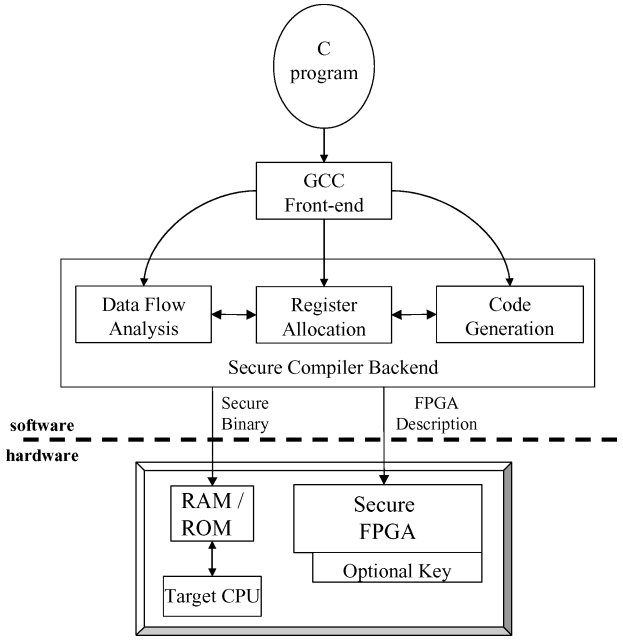


Fig. 4. SAFE-OPS experimental framework.

few instruction cycles of a typical CPU. In Section 4, this assumption is relaxed with an investigation of the performance benefits of the placement of more complex functionality on the FPGA.

### 3.3 Experimental Evaluation

As previously mentioned, in the course of allocating registers to form code sequences, the compiler will often need to insert instructions into the executable. Increasing the sequence length will have a negative impact on performance. For the scheme to work properly, the coarsest granularity of instructions that can encompass a single key is a basic block. 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, the inherent security/performance trade-offs of the SAFE-OPS approach can be evaluated by varying both the register sequence length and the percentage of encoded basic blocks.

3.3.1 *Experimental Framework.* Figure 4 shows the current SAFE-OPS experimental framework. Using a modified version of the `gcc` compiler targeting the ARM instruction set, register encoding schemes were implemented within the data-flow analysis, register allocation, and code generation phases of the `gcc` back-end. The compiler's output is (1) the encrypted application binary and (2) a description file for the secure FPGA component.

A custom hardware/software cosimulator was developed in order to obtain performance results for the following experiments. As can be seen in Figure 5,

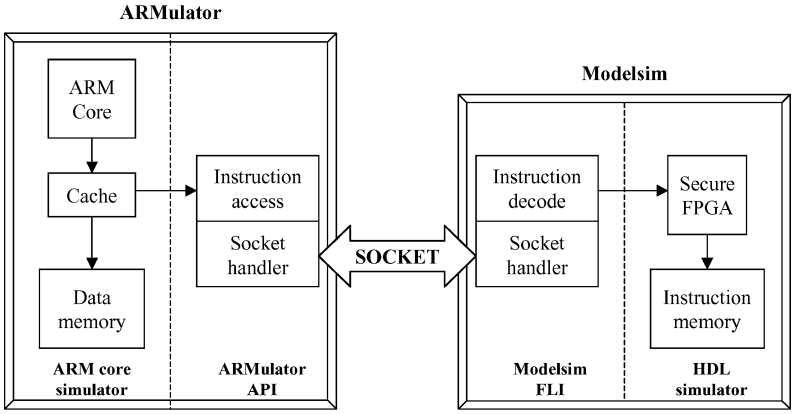


Fig. 5. HW/SW cosimulator structure.

the SAFE-OPS simulation infrastructure leverages a commercial software simulator and a commercial HDL simulator, which are connected together by a socket interface. The ARMulator simulator from ARM [1999] was utilized for modeling the software component of the system. The ARMulator is a customizable instruction set simulator which can accurately model all of the various ARM integer cores alongside memory and OS support. The ARMulator can be customized in two ways: first, by changing system configuration values (clock rate, cache size and associativity and so on), and second, through predefined modules known as the ARMulator API, that can be extended to model additional hardware or software for prototyping purposes.

For simulating the FPGA, the Modelsim simulator from Mentor Graphics [2002] was selected. Similar to the ARMulator API, the Modelsim simulator contains a foreign language interface (FLI), which allows for the simulation of C code concurrent with the VHDL or Verilog simulation. A socket interface was set up between the two simulators by using the ARMulator API and the Modelsim FLI. In the general case, when the ARMulator senses a read miss in the instruction cache, the interface code sends a signal over the shared socket to the interface written in the FLI. This signal informs the HDL simulator that an instruction read request has been made, and the appropriate FPGA code is then triggered for execution. When the FPGA finishes loading the instruction from memory and potentially decoding it, a signal is sent back through the socket to the ARMulator containing both the desired instruction along with data signifying how many execution cycles were required. Since the ARM would in general halt execution to handle the event of an instruction cache miss, the total system run-time would be equal to the sum of the total number of cycles required by the ARM for its normal execution and the total number of cycles required by the FPGA. Since the cosimulator handles the separate tasks of hardware and software simulation at a cycleaccurate level, the entire system simulation is also cycle accurate.

3.3.2 *Benchmarks.* In order to test the effectiveness of the approach, a diverse set of benchmarks from a variety of embedded benchmark suites were

Table I. Benchmarks Used for Experimentation

| Benchmark | Source | Code Size (KB) | # Basic Blocks | Instr Count (mil) |
|---|---|---|---|---|
| *adpcm* | MediaBench | 8.0 | 26 | 1.23 |
| *g721* | MediaBench | 37.9 | 84 | 8.67 |
| *arm_fir* | ARM AppsLib | 44.0 | 34 | .301 |
| *susan* | MiBench | 66.4 | 119 | 2.22 |
| *dijkstra* | MiBench | 42.5 | 32 | 7.77 |
| *fft* | MiBench | 69.2 | 44 | 4.27 |

selected. Chosen from the MediaBench [Lee et al. 1997] suite are two different voice compression programs: *adpcm*—which implements adaptive differential pulse code modulation decompression and compression algorithms, and *g721*—which implements the more mathematically complex CCITT (International Telegraph and Telephone Consultative Committee) standard. From the ARM applications library that is included with the ARM developer suite *arm_fir*—which implements a finite impulse response (FIR) filter. From the MiBench [Guthaus et al. 2001] embedded benchmark suite three applications were picked: *susan*—an image recognition package that was developed for recognizing corners and edges in magnetic resonance images (MRI) of the brain, *dijkstra*—an implementation of Dijkstra's famous algorithm for calculating shortest paths between nodes, customized versions of which can be found in network devices such as switches and routers, and *fft*—which performs a fast fourier transform (FFT) on an array of data. These benchmarks are representative of the tasks that would be required of embedded processors used in multimedia and/or networking systems. More details of the selected benchmarks can be found in Table I.

For the initial experiments, the simulator was configured to model an ARM9TDMI core which contains sixteen, 32-bit general-purpose registers. The ARM core is configured to operate at 200 MHz and is combined with separate 8 KB instruction and data caches. Setting the memory bus clock rate to be 66.7 MHz, the cache miss latency before considering the FPGA access time is 150 ns ($\sim$30 CPU cycles). This configuration is similar to that of the ARM920T processor.

The 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. However, the clock speed of an FPGA is in general determined by the critical path of its implemented logic. As will be demonstrated in Section 4, when considering architectural improvements to the FPGA decoder implementation, the potential effect on clock speed will also need to be considered.

The performance and resultant security of the SAFE-OPS approach was first explored by using this customized HW/SW cosimulator to analyze 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. The results of these experiments are presented in the following section.

3.3.3 *Register Sequence Length.* For the six selected benchmarks, the effect of increasing the encoded register sequence length on the overall system

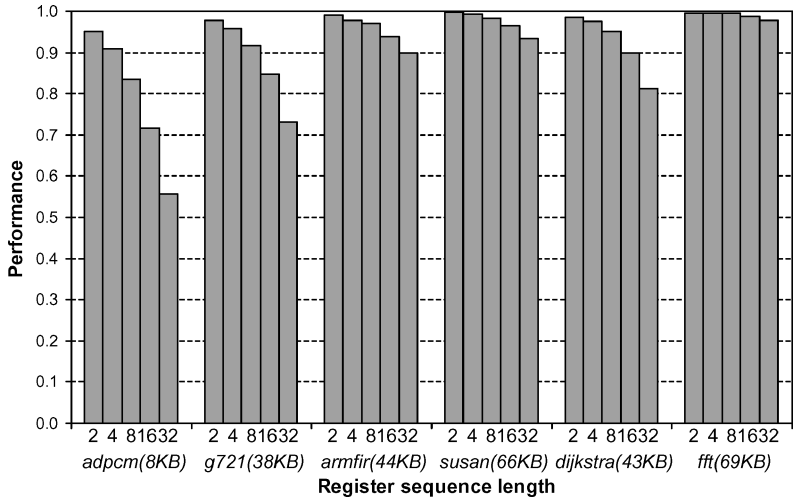


Fig. 6. Performance as a function of the register sequence length, normalized to the performance of the unencoded benchmarks. When the percentage of selected basic blocks is held constant (25%), increasing the level of security (sequence length) has a negative impact on performance. For most benchmarks, performance remained within 80% of the base case until considering the longest sequence lengths.

performance was simulated in the case when approximately 25% of the eligible basic blocks of each benchmark are encoded using a random-selection algorithm. The results, as shown in Figure 6, are normalized to the performance of the unencrypted case. As can be seen in the figure, these initial results demonstrate the potential security/performance trade-offs inherent in the SAFE-OPS approach. Overall, for most of the benchmarks the performance is within 80% of the base case (no encryption) when considering sequence lengths up to 16. However, when considering the most extreme case (when the sequence length is 32), two of the 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 explains why the two largest benchmarks, *susan* and *fft*, performed the best of the set. It is also interesting to note that two other benchmarks *arm_fir* and *dijkstra* of similar sizes do not follow the same performance trends; *arm_fir* performs almost as well as *susan* for each configuration while *dijkstra* does not. This can be explained by the fact that *dijkstra* is a much longer running benchmark, with several basic blocks with extremely high iteration counts. It is very likely that the random-selection algorithm encoded one or more or these high frequency blocks. As will be explained in the following section it is possible to select the basic blocks in a more intelligent fashion that takes these types of loop structures into account.

3.3.4 *Basic Block Selection.* In Figure 7, the case is now considered where the register sequence length is kept at a constant value of 8, and the effect of the aggressiveness of the random basic block selection technique is examined. As the upper limit on the number of encoded basic blocks is increased, it can

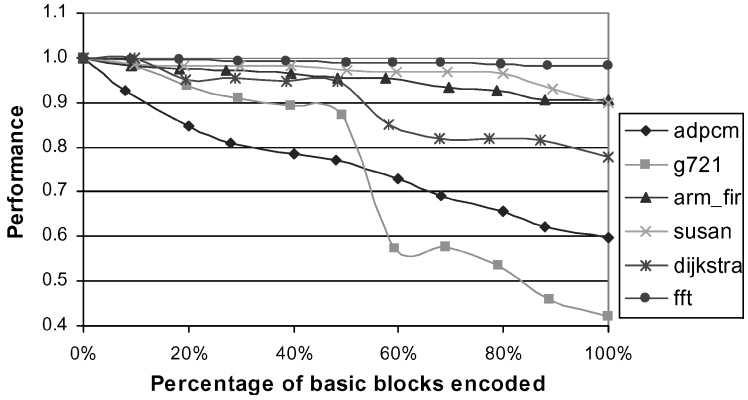


Fig. 7. Performance as a function of the rate of encoding basic blocks, normalized to the performance of the unencoded benchmarks. While for all of the benchmarks, the performance penalty when encoding less than half of the basic blocks is less than 25%, there is a potential for severe performance degradation when the overwhelming majority of the basic blocks are selected. These results motivate the development of basic block selection heuristics that take expected performance into account.

be seen that there is a limit to the performance of the resulting code. For the majority of the benchmarks, the performance in even the most secure case is within 75% of the unencrypted case. However, for the two smallest benchmarks (*adpcm* and *g721*), encrypting more than 50% of the eligible basic blocks can have a drastic effect on performance. These results show that if it is possible for the SAFE-OPS compiler to select the right basic blocks to be encoded with an appropriate sequence length value, one would be able to keep the performance at an acceptable level while still increasing security. These results motivate the development of compiler algorithms that utilize profiling and in-depth application analysis techniques in order to make better choices for selecting basic blocks. Two such approaches are discussed in the following section.

## 4. EXPLORING HARDWARE AND SOFTWARE OPTIMIZATIONS

### 4.1 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 which are most likely to be attacked—examples being basic blocks that verify for serial numbers or that check to see if a piece of software is running from its original CD. However, it is also 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.

In the previous section, a random select approach was investigated, where it was discovered that the performance penalty of the register code insertion technique was not often severe even in the case where all eligible basic blocks were encoded. However, it makes sense to develop more intelligent algorithms

for basic block selection, as this would allow the embedded application provider more flexibility in terms of positioning on the performance/security spectrum. Before developing these techniques it is necessary to first identify the root causes of the increase in total execution time.

Assuming a constant FPGA access latency, the key source of performance degradation for our approach is the addition of instructions needed 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:

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

where $\alpha^i \in [0, 1]$ is the percentage of a requested sequence 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 $\alpha^i$ value of 0.25, meaning that two 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. As an example, an encoded basic block that is executed $O(n^2)$ times will have a considerably greater performance penalty when compared to a neighboring block that is executed only $O(n)$ times. 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

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

where $CPI^i$ is the average number of cycles per instruction required of the instructions in basic block $i$. This delay value ignores the specific types of instructions inside the basic block since this variation is actually quite small. If basic block $i$ is selected for encoding using the register sequence technique, the new total execution time can be estimated as

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

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.

This can only be a partial solution, as it will be very likely that ties will exist between different basic blocks with the same $n^i_{iters}$ value. How should these blocks be ordered? The simplest method would be to order these subsets of blocks randomly, but a better approach can be found by further examining the impact on performance of the added register sequence instructions. A suitable heuristic 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

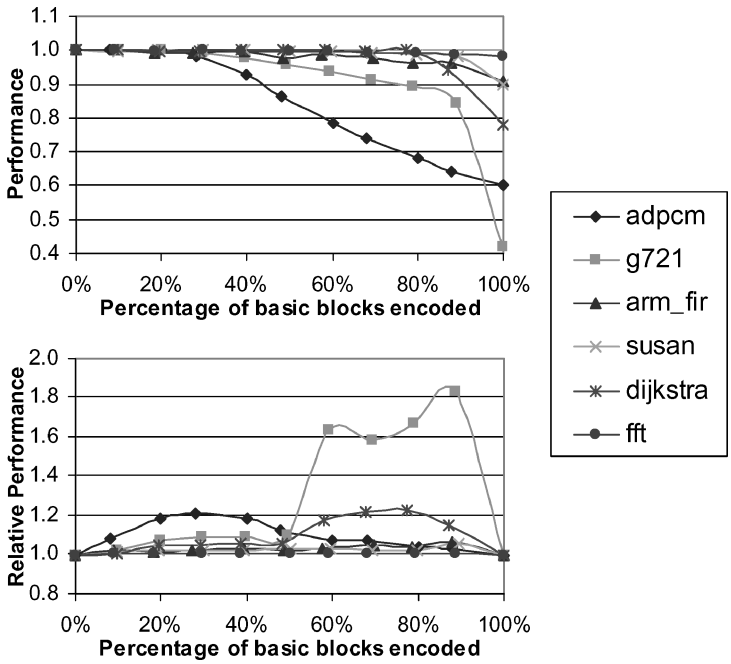


Fig. 8. Performance characteristics of the iteration-based block selection policy. The top graph shows the absolute performance impact that can be visually compared to Figure 7. The bottom graph shows the performance improvement relative to the random selection approach. These results show that by avoiding basic blocks which iterate excessively, the iteration-based policy clearly outperforms the random approach, in some cases by as much as 80%.

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 nonzero 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 SAFE-OPS 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 six 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 8 shows the general trends seen when the tests of Section 3.3.4 are rerun 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*), average 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. Conversely, the *fft* code size is relatively large, and much of the run-time is spent in mathematical library calls, instructions that are not eligible for encryption under the current approach. As is the case when no blocks are selected, when 100% of the blocks are targeted for encryption using any of the techniques the performance is the same; in these cases it is more interesting to look to architectural optimizations for a source of performance speedups.

### 4.2 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 maintaining a desirable level of security. This section investigate using the FPGA as a program-specific secondary level of instruction cache. Three specific techniques are examined:

—*L2 Cache*. As a comparison to the more application-specific caching techniques listed below, a 32-KB L2 cache was instantiated on the FPGA. As this would consume a considerable amount of programmable resources on the FPGA, this could be implemented using a relatively fast on-chip FPGA memory and therefore in this configuration the access latency requirement was increased to 8 cycles.

—*Basic Block Cache*. In this configuration, 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 decryption. This technique would gain in preference under a configuration where the FPGA decryption algorithm requires a relatively large number of computational cycles.

—*Register Sequence Buffer*. The FPGA in this architecture is able to determine which register sequence values are original instructions, and which are instructions that are inserted by the compiler to complete the sequence. This is possible with the addition of a fairly small amount of instruction decode logic. While a selected block is being fetched and validated, the FPGA stores the relative locations of the inserted register codes in a simple buffer. In future requests for these validated basic blocks, the FPGA can return NOP instructions instead of fetching instructions at these register code locations. This architecture is the simplest of the three in terms of required resources, requiring a buffer of at most $l_{key} \cdot n_{bblock}$ bits.

The common feature among these three approaches is that after the FPGA fetches an instruction from main memory and validates it, it then attempts to store the validated instruction in a faster buffer.

In evaluating these architectures, the effect on performance for three benchmarks was examined when the sequence length was kept constant at 8 and the percentage of basic blocks to be selected (randomly) was increased from 0–100%.

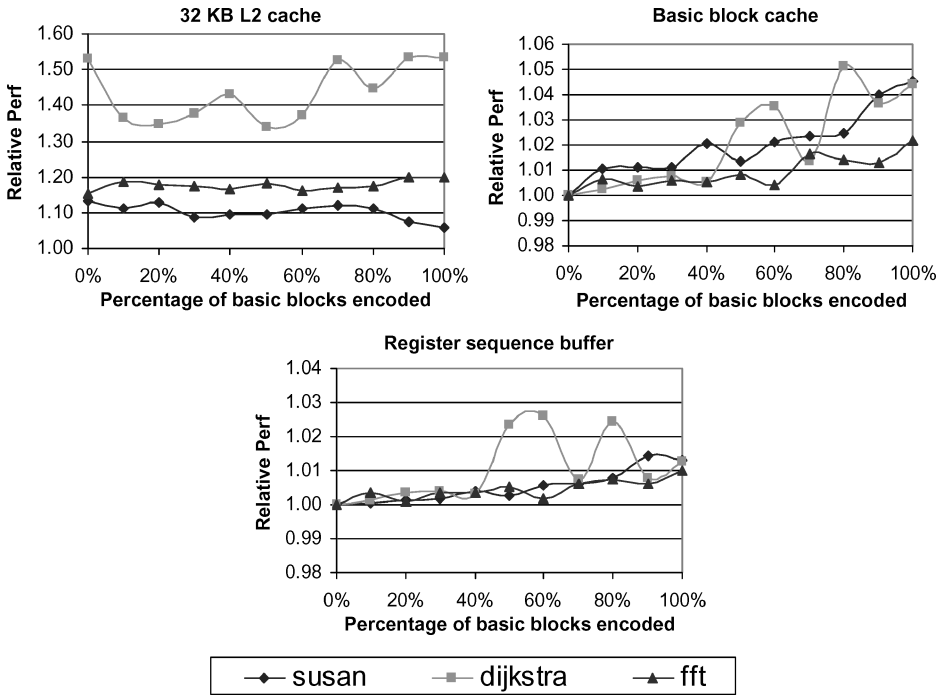


Fig. 9. Performance of the three architectural optimizations, relative to the base case when no instructions are cached in the FPGA. These results show that while the L2 cache instantiation performs by far the best of the three, the architecturally much simpler basic block caching technique slightly improves performance when the percentage of encrypted basic blocks approaches 100%.

Figure 9 shows a summary of the instruction memory hierarchy performance results of these techniques, normalized to the case where no architectural optimizations are applied. These results clearly show that the L2 cache performs much better than the other two approaches. As can be seen from the figure, instantiating an L2 cache on the FPGA leads to performance speedups of as much as 55% in some cases, as compared to the basic block caching approach which is limited to a 5% speedup, and the register sequence buffer which demonstrates less than 3% performance improvement. This is to be expected, as the relatively large cache allows us to buffer a high percentage of all instructions in the benchmarks. While the L2 cache technique performs the best overall, it is interesting to note that the basic block cache and register sequence buffer approaches show near-linear speedups as the aggressiveness of the encryption is increased. This is an important result, as it implies that if the other system parameters are tuned for more cryptographic strength (i.e., by increasing the key length, implementing more complex instruction filters, implementing more algorithmically advanced register sequence translators and others), these two software protection-specific approaches will begin to perform favorably well when compared to the general L2 cache case.

When evaluating the implementation of architectural-based optimizations on the FPGA it is necessary to consider that the clock speed of the FPGA is

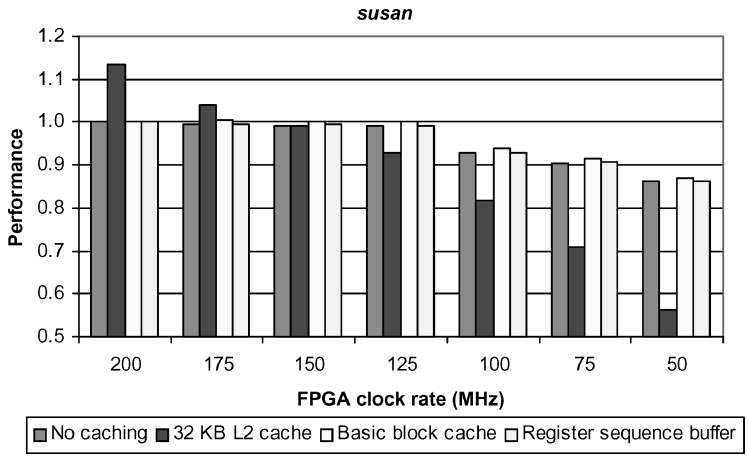


Fig. 10. Performance of the three architectural caching strategies for varying FPGA clock rates. These results demonstrate that the resource-utilization of the FPGA needs to be taken into account when considering architectural optimizations, as the strategy that performs the best under the assumption of an ideal clock rate (L2 cache), ultimately performs the worst given the condition that the clock rate would need to be lowered to 50 MHz to compensate for the additional applied logic.

dependent on the critical path of the instantiated logic. In order to effectively measure the performance of an FPGA architectural technique, it is important to consider configurations of varying clock speeds. Figure 10 shows the results of such experiments on the *susan* benchmark. In gathering these results, the register sequence length was held constant at 8 and the percentage of encoded basic blocks constant at 25%. From this figure, it can be clearly seen that as the clock rate of the FPGA is reduced via simulation the performance benefits of the L2 cache implementation can quickly turn negative. This drastic effect on instruction memory hierarchy performance is a function of the additional FPGA cycles required by the architecture. This is due to the fact that the memory bus speed is kept constant, and any cycles spent by the FPGA grow in importance as its clock rate decreases. These results show that if the L2 cache implementation required a clock rate of 150 MHz or slower, the basic block cache and register sequence buffer techniques would perform better, assuming their minimal logic would allow them to continue operating at 200 MHz.

## 5. CONCLUSIONS AND FUTURE WORK

The rapidly increasing use of embedded processors in consumer, telecommunications, and transportation applications, along with the recent trend of including networking capabilities in even the simplest of these devices, motivates the need for an intensified scrutiny of the security of system software. In this paper, we presented a novel approach to embedded software protection that utilizes a combination of compiler and architecture techniques. The main benefit of this approach is the flexibility it allows the embedded application designer in terms of positioning on the security/performance spectrum. Also, the framework is entirely compatible with the current purely software approaches to

software protection. The presented results show that, when coupling register sequencing compiler algorithms with optimized FPGA architectures, the overall application security was improved at only a nominal cost to performance.

Several improvements to the SAFE-OPS framework are being considered. The current compiler output is an encrypted application binary and a description file for the secure FPGA. While this description file currently contains code for configuration, in the future the compiler will be able to generate a customized behavioral specification of the FPGA decoding and decryption algorithms in a hardware description language such as VHDL or Verilog. Also, it would be useful for the embedded application designer to be able to assist the compiler in selecting which segments of code or data are to be encrypted. This requires some front-end additions to the compiler framework for the inclusion of user directives. Finally, this current approach operates only on source code. This creates difficulties for the developer who wants to secure an entire application, but is dependent on statically compiled 3rd-party libraries for the generation of executables. Future tools will utilize decompilation techniques to enable this protection approach to be applied to pre-compiled code.

## ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their valuable feedback on the earlier drafts of this paper.

## REFERENCES

ACTEL. 2003a. CoreDES data sheet, v2.0. Available at http://www.actel.com.

ACTEL. 2003b. Design security with Actel FPGAs. Available at http://www.actel.com.

APPEL, A. W. AND FELTEN, E. W. 1999. Proof-carrying authentication. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*. 52–62.

ARM. 1999. Application note 32: The ARMulator. Available at http://www.arm.com.

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

BAIFANZ, D., DEAN, D., AND SPREITZER, M. 2000. A security infrastructure for distributed Java applications. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*. 15–26.

BARAK, B., GOLDREICH, O., IMPAGLIAZZO, R., RUDICH, S., SAHAI, A., VADHAN, S., AND YANG, K. 2001. On the (im)possibility of obfuscating programs. In *Proceedings of Advances in Cryptology (CRYPTO '01)*. 1–18.

BAUER, L., SCHNEIDER, M., AND FELTEN, E. 2001. *A Proof-Carrying Authorization System*. Tech. Rep. CS-TR-638-01, Department of Computer Science, Princeton University.

BELLARE, M., CANETTI, R., AND KRAWCZYK, H. 1996. Message authentication using hash functions: The HMAC construction. *RSA Laboratories' CryptoBytes 2*, 1 (Spring).

CHANG, H. AND ATALLAH, M. 2000. Protecting software code by guards. In *Proceedings of the ACM Workshop on Security and Privacy in Digital Rights Management*. 160–175.

CHANG, S., LITVA, P., AND MAIN, A. 2001. Trusting DRM software. In *Proceedings of the W3C Workshop on Digital Rights Management for the Web*.

CLARK, P. AND HOFFMAN, L. 1994. BITS: A smartcard protected operating system. *Commun. ACM 37*, 11 (Nov.), 66–70.

COLLBERG, C. AND THOMBORSON, C. 1999. Software watermarking: models and dynamic embeddings. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 311–324.

COLLBERG, C. AND THOMBORSON, C. 2002. Watermarking, tamper-proofing, obfuscation: Tools for software protection. *IEEE Trans. Software Eng. 28*, 8 (Aug.), 735–746.

COLLBERG, C., THOMBORSON, C., AND LOW, D. 1997. *A Taxonomy of Obfuscating Transformations*. Tech. Rep. 148, Department of Computer Science, The University of Auckland.

COLLBERG, C., THOMBORSON, C., AND LOW, D. 1998. Breaking abstractions and unstructuring data structures. In *Proceedings of the IEEE International Conference on Computer Languages*. 28–38.

COMPUTER SECURITY INSTITUTE AND FEDERAL BUREAU OF INVESTIGATION. 2002. CSI/FBI 2002 computer crime and security survey. Available at http://www.gocsi.com.

DAEMAN, J. AND RIJMEN, V. 2000. The block cipher Rijndael. In *Smart Card Research and Applications*, J.-J. Quisquater and B. Schneier, Eds. Lecture Notes in Computer Science, vol. 1820. Springer-Verlag, Berlin, 288–296.

DALLAS SEMICONDUCTOR. 1999. Features, advantages, and benefits of button-based security. Available at http://www.ibutton.com.

DANDALIS, A., PRASANNA, V., AND ROLIM, J. 2000. An adaptive cryptographic engine for IPSec architectures. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. 132–144.

FISHER, M. 2000. Protecting binary executables. *Embedded Syst. Program. 13*, 2 (Feb.).

GOBIOFF, H., SMITH, S., TYGAR, D., AND YEE, B. 1996. Smart cards in hostile environments. In *Proceedings of the 2nd USENIX Workshop on Electronic Commerce*. 23–28.

GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th IEEE Annual Workshop on Workload Characterization*. 3–14.

HOFFMANN, A., MEYR, H., AND LEUPERS, R. 2002. *Architecture Exploration for Embedded Processors Using LISA*. Kluwer Academic Publishers, Boston, MA.

HORNE, B., MATHESON, L., SHEEHAN, C., AND TARJAN, R. 2001. Dynamic self-checking techniques for improved tamper resistance. In *ACM Workshop on Security and Privacy in Digital Rights Management*. 141–159.

IBM. 2002. Secure systems and smart cards. Available at http://www.research.ibm.com/secure_systems.

ITOI, N. 2000. *Secure Coprocessor Integration with Kerberos* V5. Tech. Rep. RC-21797.

JOHNSON, N. AND KATZENBEISSER, S. 1999. *A Survey of Steganographic Techniques*. Artech House, Boston, MA.

KAPS, J.-P. AND PAAR, C. 1998. Fast DES implementations for FPGAs and its application to a universal keysearch machine. In *Proceedings of the 5th Annual Workshop on Selected Areas in Cryptography*. 234–247.

KOCHER, P. 1995. Cryptanalysis of Diffie-Hellman, RSA, DSS and other systems using timing attacks. Extended abstract.

KOMMERLING, O. AND KUHN, M. 1999. Design principles for tamper-resistant smartcard processors. In *Proceedings of the USENIX Workshop on Smartcard Technology*. 9–20.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of 30th Annual International Symposium on Microarchitecture*. 330–335.

LIE, D., THEKKATH, C., MITCHELL, M., LINCOLN, P., BONEH, D., MITCHELL, J., AND HOROWITZ, M. 2000. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*. 168–177.

MENTOR GRAPHICS. 2002. Modelsim SE simulation and verification datasheet. Available at http://www.mentor.com.

NECULA, G. 1997. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 106–119.

NECULA, G. 2003. Web page http://www.cs.berkeley.edu/ñekula.

NECULA, G. AND LEE, P. 1996. Safe kernel extensions without run-time checking. In *Proceedings of the 2nd USENIX Symposium on OS Design and Implementation*. 229–243.

NEVE, A., FLANDRE, D., AND QUISQUATER, J.-J. 1999. Feasibility of smart cards in Silicon-on-insulator (SOI) technology. In *Proceedings of the USENIX Workshop on Smartcard Technology*.

PRASANNA, V. AND DANDALIS, A. 2000. FPGA-based cryptography for internet security. In *Online Symposium for Electronic Engineers*.

SCHNEIER, B. AND SHOSTACK, A. 1999. Breaking up is hard to do: modeling security threats for smart cards. In *Proceedings of the USENIX Workshop on Smartcard Technology*. 175–185.

SMITH, S. 1996. *Secure Coprocessing Applications and Research Issues*. Tech. Rep. LA-UR-96-2805.

SMITH, S. AND WEINGART, S. 1999. Building a high-performance programmable secure coprocessor. *Comput. Networks 31*, 9 (Apr.), 831–860.

TAYLOR, R. R. AND GOLDSTEIN, S. 1999. A high-performance flexible architecture for cryptography. In *Proceedings of the Workshop on Cryptographic Hardware and Software Systems*.

TRUSTED COMPUTING PLATFORM ALLIANCE. 2003. http://www.trustedcomputing.org.

TYGAR, D. AND YEE, B. 1993. Dyad: A system for using physically secure coprocessors. In *Proceedings of the Harvard-MIT Workshop on Protection of Intellectual Property*.

WANG, C., DAVIDSON, J., HILL, J., AND KNIGHT, J. 2001. Protection of software-based survivability mechanisms. In *Proceedings of the 2001 IEEE/IFIP International Conference on Dependable Systems and Networks*.

WANG, C., HILL, J., KNIGHT, J., AND DAVIDSON, J. 2000. *Software Tamper Resistance: Obstructing the Static Analysis of Programs*. Tech. Rep. CS-2000-12.

WEINGART, S. 1987. Physical security for the mABYSS system. In *Proceedings of the IEEE Symposium on Security and Privacy*. 52–58.

WEINGART, S., WHITE, S., ARNOLD, W., AND DOUBLE, G. 1990. An evaluation system for the physical security of computing systems. In *Proceedings of the 6th Computer Security Applications Conference*. 232–243.

WHITE, S. AND COMERFORD, L. 1987. ABYSS: A trusted architecture for software protection. In *Proceedings of the IEEE Symposium on Security and Privacy*. 38–51.

WHITE, S., WEINGART, S., ARNOLD, W., AND PALMER, E. 1991. *Introduction to the Citadel Architecture: Security in physically Exposed Environments*. Tech. Rep. RC 16682.

WYANT, J. 2001. Establishing security requirements for more effective and scalable DRM solutions. In *Proceedings of the Workshop on Digital Rights Management for the Web*.

YEE, B. 1994. *Using Secure Coprocessors*. Tech. Rep. CMU-CS-94-149.

YEE, B. AND TYGAR, D. 1995. Secure coprocessors in electronic commerce applications. In *Proceedings of the 1st USENIX Workshop on Electronic Commerce*. 155–170.

Received March 2003; revised October 2003, April 2004; accepted May 2004