# Enabling Trusted Software Integrity

Darko Kirovski
Microsoft Research
One Microsoft Way
Redmond, WA 98052
darkok@microsoft.com

Milenko Drinić
Computer Science Dept.
University of California
Los Angeles, CA 90095
milenko@cs.ucla.edu

Miodrag Potkonjak
Computer Science Dept.
University of California
Los Angeles, CA 90095
miodrag@cs.ucla.edu

## ABSTRACT

Preventing execution of unauthorized software on a given computer plays a pivotal role in system security. The key problem is that although a program at the beginning of its execution can be verified as authentic, while running, its execution flow can be redirected to externally injected malicious code using, for example, a buffer overflow exploit. Existing techniques address this problem by trying to detect the intrusion at run-time or by formally verifying that the software is not prone to a particular attack.

We take a radically different approach to this problem. We aim at intrusion prevention as the core technology for enabling secure computing systems. Intrusion prevention systems force an adversary to solve a computationally hard task in order to create a binary that can be executed on a given machine. In this paper, we present an exemplary system —SPEF— a combination of architectural and compilation techniques that ensure software integrity at run-time. SPEF embeds encrypted, processor-specific constraints into each block of instructions at software installation time and then verifies their existence at run-time. Thus, the processor can execute only properly installed programs, which makes installation the only system gate that needs to be protected. We have designed a SPEF prototype based on the ARM instruction set and validated its impact on security and performance using the MediaBench suite of applications.

## 1. INTRODUCTION

Preventing execution of unauthorized software on a computing system is an essential part of system security. Most computing systems rely on the operating system and basic cryptographic primitives to provide security features that ensure data, program, and execution flow authenticity and integrity. Unfortunately, the complexity of modern operating systems (e.g., Microsoft's Windows XP Embedded contains up to 10,000 integrated modules) and the fact that an adversary needs only a single unprotected entry point to gain control over a system, have made malicious code into a common security problem on all systems that allow incoming traffic from distrusted sources such as the Internet.

A key problem in such systems is that although a program at the beginning of execution may be verified as authentic, while running, its execution flow can be redirected to externally injected malicious code using, for example, a buffer overflow exploit [22]. Once the adversary executes injected code in the highest trust-priority mode, usually all system resources are at her disposal. In that case, the possibility for malicious actions is fairly broad including: destruction (e.g., disk formatting, deleting files), replication (e.g., Internet worms), network traffic analysis (e.g., packet sniffing), and covert communication (e.g., Trojan horses). Ease of implementation and effectiveness have established attacks that focus on redirecting program execution as the most common threat to system security [18, 6]. The research community has addressed this problem from two perspectives:

- *Intrusion detection* – a set of mechanisms that aim at scanning system resources and detecting the activity of intrusive agents [5].
- *Formal verification* – a set of formally defined methods that either change the definition of the programming language so that executables are impervious to buffer overflow attacks [12], or perform static analysis on binaries to verify that they do not have buffer overflow exploits [27].

### 1.1 Intrusion Prevention

In this paper, we take a radically different approach. We aim at **intrusion prevention** as the core technology for enabling secure computing systems. *The goal of an intrusion prevention system is to force an adversary to solve a computationally difficult (preferably intractable) task in order to create a binary that can execute on a given machine.* According to this definition, the only way that an adversary can inject software into a protected system is by forcing a trusted user to explicitly install it, which is, in a sense, the best we can hope for, as a trusted user can always choose to install a malicious program.

As a demonstration of an intrusion prevention system, in this manuscript, we introduce —**SPEF**[1]— a framework of architectural and compilation mechanisms that ensure software integrity at run-time. SPEF installs a software binary by encoding a set of constraints, unique to the processor, into each block of instructions. Specifically, the binary is modified using local peephole transformations and

[1]SPEF – **S**ecure **P**rogram **E**xecution **F**ramework.


*ASPLOS X* 10/02 San Jose, CA, USA

rescheduling techniques that do not alter program functionality and that have nominal impact on code run-time. SPEF exploits the fact that a given program binary has numerous representations that all satisfy the desired functionality and which all are within certain small performance bounds. The entropy of these representations is used as a communication channel to store the processor-specific constraints.

While constraint encoding is a post-compilation procedure, constraint verification is performed in the processor for each block of referenced instructions. The difficulty of creating a valid block of instructions stems from the fact that the constraints are encoded using a keyed and cryptographically secure one-way hash function, where the secret key is hidden in the processor hardware and cannot be accessed by any program except the software installer. Thus, software installation must be performed in a special mode, much like the boot procedure on a modern PC.

## 1.2 System Features Enabled by SPEF

SPEF is a cryptographic tool used by the operating system to ensure that users are not able to switch from their user-mode into another one that is either of higher security or that impersonates another user. SPEF represents a fundament for secure execution of programs as it can police the crucial three trust modes on a given platform: *trusted OS kernel mode* which cannot be accessed by any user, all variants of *user modes* (private, public, shared), and a distrusted *public mode*. Hence, SPEF can actually provide a platform for execution of an arbitrary distrusted program, a scenario typical for distributed computing platforms such as peer-to-peer networks. SPEF enables the operating system to realize the following basic modes of operation.

PUBLIC mode (a) – in this mode, SPEF is disabled. Therefore, this mode does not provide protection for software integrity. On the other hand, since SPEF is not used in this mode, the user can run performance-critical applications that do not require software protection. Similarly, the user can run programs downloaded from the Internet that have not being authenticated, while having the insurance that these programs cannot launch a buffer overflow or similar code alternation attack against processes running at a higher priority. Finally, this mode opens a spectrum of opportunities for deploying a computer in a peer-to-peer networking scenario: including file-sharing, collaboration, etc. Just as in a traditional system, the computer resources available in this mode are governed by the administrator – either UNIX-like `chmod` type of access control or a digitally signed registry are both possible as a control mechanism.

TRUSTED mode (b) – in this mode, SPEF is enabled which means that all programs executed in this mode must be installed. Since programs in this mode are verified for integrity and authenticity, SPEF can protect system and user resources from malicious users. Trusting the operating system is important from the perspective that the administrator user (e.g., the most privileged user) must not be able to access the resources that are protected for another user if sharing is not specified. In that respect, there can be two trusted submodes:

(b.1) USER MODE – where information and software integrity are protected among users with typical submodes such as private and shared resources.

(b.2) OS KERNEL MODE – this mode is the root of trust for the maintenance of security policies among users. Every time a resource is accessed, the operating system takes over the call in this mode, to ensure that proper access is given to accredited entities. This mode can be enforced using several mechanisms – the simplest one probably being fixed memory mapping. The description of the environment that would handle such policing is beyond the scope of this paper, however, it strongly resembles traditional operating systems such as UNIX or Microsoft Windows.

INSTALLATION mode (c) – the purpose of installation in SPEF is to encode processor specific constraints into programs such that they can be executed in trusted mode. Since the constraints are dependent upon the root of system security, processor's secret key, the requirement imposed upon a process executed in installation mode is not to leak this secret under software and most hardware attacks. Details of how this mode operates are presented in Subsection 3.2.

The operating system can run in modes (a-b), however, its access management procedures must run in the (b.2) mode. Since SPEF induces certain performance overhead, in order to trade speed for security, users can run programs in mode (a) for maximum performance. Clearly, SPEF enables flexibility in terms of functionality, performance, and strong software integrity of the computing platform.

# 2. ATTACK MODEL

In this paper, we consider a fairly broad class of attacks in which the goal of the adversary is to subvert the execution of a given program into a program of adversary's choice at the highest trust-priority mode. In such a scenario, the adversary operates either remotely (typical for client-server scenarios) or from a lower trust-priority mode (typical for peer-to-peer networks or application service providers). In the first case, the adversary scans the remote computer's ports for networking services with known security flaws and then penetrates the system by exploring their vulnerabilities. In the second case, the adversary is already running a program on the remote system, but not at the highest trust-priority mode. In this case, the adversary can use the vulnerability of any system procedure already running in the top priority mode and try to subvert its execution flow towards a desired malicious procedure. System vulnerabilities usually stem from network and cryptographic protocol incorrectness (such as the 802.11 hack [2]), careless protocol or system call implementations (e.g., the `setuid()` system calls on UNIX systems [4]), or improperly conceived I/O.

In this section, we focus on the most common type of attacks that belong to this class: buffer overflows[2]. In order to be vulnerable to this attack, a program must have at least one buffer with an unchecked overflow. The simplest buffer overflow attack, stack smashing [22, 18], overwrites a buffer on the stack to replace the function call return address. Figure 1 illustrates how execution is subverted in the case of stack smashing. The adversary writes into the target buffer enough information to overwrite the return address of the currently executed procedure and then places the malicious code. The return address is overwritten with the

[2]Also referred to as buffer overruns.

address of the injected malicious procedure. The malicious code is typically a call to a function such as `execve()` on UNIX systems which kills the current process and launches an arbitrary binary in the same trust priority mode as was the killed process. Needless to say, the attack has numerous variants tailored to particular operating systems or programming language features.

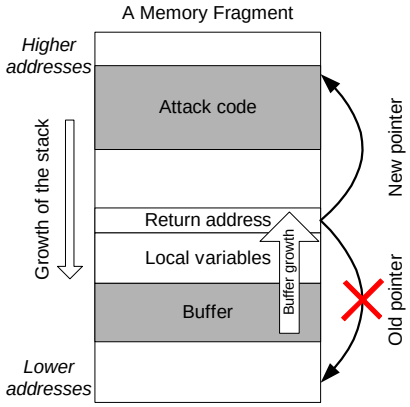


**Figure 1: Buffer overflow attack. By overfilling the unchecked buffer, the attacker overwrites the original return address of the currently executed procedure and injects afterwards the malicious procedure. Upon return from the currently executed procedure, control flow is diverted to the malicious code.**

Buffers with unchecked overflow commonly occur in software for two main reasons: most common programming languages do not inherently generate buffer manipulation code that checks for overflows and the inevitable programmer's mistakes. For example, programs written in C, which provides direct low-level memory access and pointer arithmetic, are particularly susceptible to buffer overflow attacks as common standard string functions (e.g., `gets()` or `strcpy()`) do not check for buffer overflow.

The solutions to buffer overflow attacks can be divided into two categories: run-time detection and static source code analysis. A typical example of the first type of solutions is StackGuard, a set of compiler enhancement mechanisms, that generates binaries which insert a "dummy" value on the stack next to the return address [5]. Programs check if the value has been tampered with before jumping back to the calling function. The technology provides decent protection against most common buffer overflow attacks with a relatively large performance overhead [5]; however, the technology is not provably secure against a generic buffer overflow attack [5]. Similar, run-time solutions observe potentially dangerous operations to restrict further execution in case of an intrusion [3, 9].

Static source code analysis techniques range from fast and simple error detection, such as type errors and uninitialized variables [11, 28] to complex, powerful, and relatively slow formal verification-based tools that detect variety of bugs, including null pointers and errors in definitions, allocation and aliasing [8, 7, 24]. Wagner et al. used static analysis techniques as the first step toward automated buffer overflow detection – their static analyzer generates large number of false alarms that must be verified manually [27].

In general, both run-time intrusion detection and static analysis including formal verification may prove to be quite effective in certain cases. However, both approaches require manual support and hence fall short in the case of a human error, the most prolific vulnerability manufacturer. The possibilities for intrusion are numerous – e.g., every floating pointer in a program can be used potentially as an intrusion gate. The methodology that we present in this work aims at preventing an arbitrary intrusion attack; we aim at forcing the adversary solve a difficult problem rather than having the intrusion detection or formal verification software perform a computationally challenging task in the pursuit for impenetrable system security.

## 3. SPEF - SYSTEM DESCRIPTION

The main goal of SPEF is to ensure that a potential intruder has to solve a computationally intractable task in order to execute any desired code on a SPEF-protected machine. SPEF achieves this objective by relying on both hardware and software-based mechanisms. A block of instructions[3] is an atomic execution unit on a SPEF machine. During software installation, each block of instructions is transformed depending upon a cryptographic key. This key is securely stored in the processor. The transformations are invariant with respect to functionality. During the transformation, SPEF first computes a transformation-invariant hash of the instruction block. In other words, the hash is such that regardless of the considered instruction transformations (e.g., instruction rescheduling), it always has the same value. The computed hash is then encrypted with the secret key, which results in a random string of bits. The values of these bits are used to select a unique transformation of the instruction block (e.g., a particular instruction order in the case of instruction rescheduling).

During program execution, the SPEF verifier, part of processor's hardware, repeats the steps taken by the software installation procedure. First, it computes the hash of the instruction block. Note that because the hash value is invariant to the applied transformations, the hash value computed during verification is equal to the one computed during installation. Then, it generates the random string of bits. Finally, it verifies that the instruction block satisfies the constraints imposed by this bitstream (e.g., instruction order is the same as the order imposed by the computed bitstream). If the constraints are not satisfied, the SPEF verifier concludes that the instruction block has not been installed properly or that it has been altered and generates an interrupt that aborts the execution of the current process.

Intuitively, SPEF is a cryptographic tool that signs a block of instructions during installation and stores this signature in the instruction block itself. The signature (random bitstream) does not need to use public-key cryptography because the signing key (processor's ID) is never revealed externally. In an inefficient implementation of SPEF, the signature can be stored in the instruction block as is, for example, as a footer. However, cryptographically secure hashes such as SHA1 or MD5 require at least 20 bytes of storage space, which increases code size by an excessive amount. We exploit the fact that a given program binary has numerous representations that all satisfy the desired functionality and which all are within certain small performance bounds. The

[3]Depending on the implementation, the length of this block can correspond, for example, to the size of the instruction cache line or pre-fetch buffer.

entropy of these representations is used as a communication channel to store the signature, which significantly reduces both the storage and, indirectly, performance overhead imposed by the embedded signature.

The most important potential limitation of SPEF is its performance overhead – a price that needs to be paid for trustworthy computing. First of all, as experiments show, this overhead is reasonably low. In addition, SPEF is not conceived as a mandatory execution platform within a computing system – it can be deployed only for select processes that require high level of software and data integrity (e.g., OS kernel, e-mail agents). Processes that do not require such level of security can be executed in a mode that does not deploy SPEF run-time code verification and hence, gets the maximum performance from the system.

## 3.1 Preliminaries

SPEF is a platform that consists of architectural mechanisms and compilation tools with intrusion prevention as an objective. In this section, we present the key entities in SPEF and describe their interaction during software installation and run-time software verification. SPEF's key components and their activity are illustrated in Figure 2.

PROCESSOR-UNIQUE IDENTIFIER - CPU ID. The root of security is a read-only register with a secret key that is unique for each processor. Burning unique identifiers is a standard manufacturing practice as demonstrated by the Pentium III processors [10]. Intel's goal with the Processor Serial Number (PSN) was to provide a link between computers and their "owners", then identify computers to external agents (such as e-commerce web-sites), who can in return develop services for their "owners" tailored to this feature. Such an application posed a myriad of privacy and impersonation concerns that forced Intel to disable this feature on their products[29].

The main goal of SPEF is to secure privacy rather to expose identifying information to external agents. Hence, the privacy problem does not occur in SPEF systems for two reasons: ($a$) the information that links SPEF's CPU ID with its "owner" is not maintained just as with any other chip on the market with a serial number and ($b$) as a root of system security, the CPU ID is never revealed to any installed application, let alone an external agent[4]. Thus, it is in the best possible interest to the designers of a SPEF-like system to keep the CPU identifier as secret as possible.

SOFTWARE DELIVERY. There are two essential procedures associated with software delivery: initial software installation and software updates, e.g., patching. In a SPEF-based system, from the perspective of a software distributor, these two actions are exactly the same as in conventional computing systems. A given application is distributed to clients or peers as a compiled binary - we denote it as the **master-copy**. The recipients of the software can validate the authenticity of the master-copy via standard authentication methods that deploy public-key cryptography (e.g., via SSL). The master-copy cannot be executed on any of the target machines. Each target machine needs to customize the master-copy in order for it to run. The installed copy of the program, denoted as the **working-copy**, is functionally equivalent to the master-copy but augmented with processor-specific constraints. In high-security mode, only a working-copy can be run on the SPEF platform. The same installation procedure needs to be performed for both the initial installation and subsequent updates.

[4]One remote possibility for an adversary who has physical access to a SPEF processor, is to extract/alter CPU's ID by reverse engineering, and then perform malicious activities accordingly. The cost of such an attack is significant as an adversary must disassemble, reverse engineer, and reassemble a SPEF-like chip without destruction.

## 3.2 Software Installation

Software installation consists of several steps illustrated in Figure 2. In the first step, the CPU enters a special **installation mode** used exclusively for software installation. Since the CPU ID is a system's master secret, it must never leave the pins of the CPU. The installer must access the CPU ID to create application's working-copy. This special mode of operation ensures that the CPU ID is kept secret from software and most hardware attacks. This is done by following a simple recipe: ($a$) installation is executed atomically, i.e. without any interruptions, ($b$) the installer must not write the CPU ID or any other variable that discloses bits of the CPU ID off chip, and ($c$) before completion, the installation procedure must overwrite any intermediate results or variables stored in on-chip memory. Here is an example of steps that a CPU should follow to enter this mode:

CALLING THE INSTALLER. First, the arguments to the software installer are fetched to a predetermined and fixed address in the main memory. The arguments encompass the location and size of the input master-copy, the location where the working-copy should be stored, and potentially, a password[5] that enables only an authorized party to install software. Following the principles of traditional password systems, the hash of this password is stored on-chip in non-volatile memory (e.g., flash memory). Before proceeding to the next step, the CPU verifies whether the hash of the entered password is the same as the stored hash and then continues to the next step.

Since the installer operates in a single process mode, it uses physical rather than virtual addresses. Thus, the OS needs to forward the physical addresses of source and destination buffers to the installer. This can be done in several ways that may include hardware support. The OS is responsible for freeing the operating memory used as an output destination. If the OS is not installed, the installer assumes certain default values for its arguments.

DISABLING CONTEXT SWITCHING. The calling procedure must disable all soft and hard interrupts on the CPU before issuing the call. Hardware can verify that CPU ID is accessed only if all interrupts are disabled. Alternatively, the CPU can have hardware support to buffer interrupts while in the installation mode. This renders context switching on the CPU during software installation impossible. This step is important for preventing possible leakage of secrets or other types of manipulations.

DIVERTING CONTROL TO THE INSTALLER. Finally, the program counter is redirected to the memory address where the software installer is located. The software installer is a program that can be stored on-chip in read-only memory or

[5]User authentication can be also achieved using smart cards.

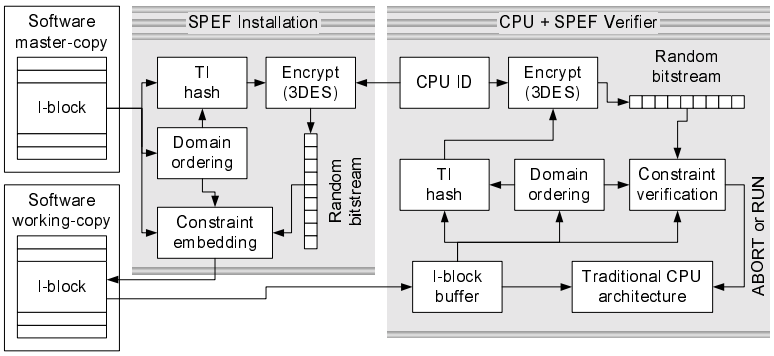


**Figure 2: Flowchart describing the components and procedures of the SPEF system.**

externally. In the latter case, the content of the external ROM must be validated using a cryptographic hash within the CPU prior to loading the first instruction of the installer. In this case, the expected value of the hash must be stored permanently[6] in the CPU.

Note that this mode of operation is actually not particularly unique, as most boot procedures for modern processors are performed in similar fashion from external ROMs containing BIOS-like software [19, 16].

### 3.2.1 Constraint Encoding

The main goal of the software installer is to encode unique constraints for a given processor, into the target binary in a secure manner. An application's working-copy is created by processing individual blocks of instructions. An instruction block (I-block) is an atomic execution unit; even if the CPU executes only a single instruction from an I-block, the entire I-block must be validated to begin execution of this instruction. I-block size is governed by the size of the instruction pre-fetch buffer in the CPU - typically block size is 64 to 256 instructions. Smaller blocks do not offer sufficient transformation entropy, while larger blocks impose hardware and performance overhead.

Installation of each I-block is performed in three steps: domain ordering, computation of the transformation-invariant hash of the I-block, and finally, constraint embedding. SPEF can operate with different types of constraint encoding techniques such as: assigning variables to a pseudo-random permutation of available registers, instruction scheduling, branch-type selection, and basic block reordering. Note that all these transformations preserve program's functionality. For the sake of brevity, here we use instruction rescheduling as an exemplary constraint to demonstrate the procedures involved in software installation (details on constraint encoding are presented in Section 4.1). The installation process of an I-block is illustrated in Figure 3.

Domain ordering. For a given type of transformation, the goal of domain ordering is to assign a unique identifier to each component of the domain in which constraints are embedded. The ordering policy must be invariant under the transformations used in constraint encoding, so the assigned unique identifiers are identical before and after constraint embedding. For example, domain ordering for instruction scheduling could be to assign a unique number to each instruction within a basic block according to a standardized policy (e.g., sorting instructions with preserved dependencies based on their op-code).

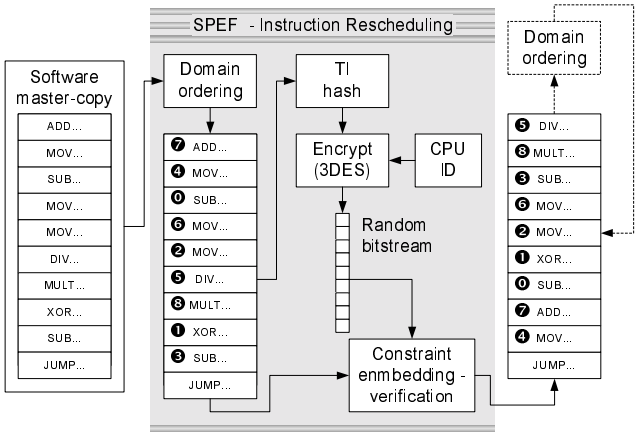


**Figure 3: Constraint encoding on the example of instruction rescheduling as a transformation.**

Transformation-invariant hash (TI-hash). In this step, SPEF computes the TI-hash of the content of the I-block. The TI-hash is a sequence of bits used to generate the constraints. Clearly, since the embedded constraints must be invariant under the change that they introduce in the I-block, the TI-hash must be invariant with respect to the transformations that embed the constraints. Because of the transformation invariance, the TI-hashes computed for the master- and working-copy of an I-block are equal. As a direct consequence, any working-copy from an arbitrary client can be used as a master-copy on all other clients. An example of structures that stay intact after common exemplary transformations, are: control-data flow graphs, instruction types, and values of constants.

Constraint embedding. Constraints are embedded into the I-block as directed by a 256-bit pseudo-random bitstream. This stream is created by encrypting the TI-hash using a standard cipher (3DES or AES) keyed with the secret CPU ID. For example, for instruction scheduling, embedding a set of constraints means that a particular schedule

[6]Replacement of this hash value may be allowed only in installation mode. Then, the CPU must store on-chip the public key of the trusted authority issuing valid hash values. This key is used to verify signatures of new hash values.

of instructions within an I-block is enforced with preserved functional dependencies. A single constraint is encoded as the order in which two independent instructions `I0` and `I1` appear (e.g., `0` from the bit-stream directs `I0` to appear before `I1`). The modified I-blocks are assembled into a final working-copy of the executable.

## 3.3 Program Execution

The working-copy of a program can be executed on the processor in any of the operation modes enabled by SPEF. While executing the working-copy of the program, the processor performs several steps: it loads a block of instructions, verifies the embedded constraints in this I-block, and finally approves its execution. The verification procedure consists of the same three steps as constraint encoding. The only difference is that instead of embedding constraints, the verifier analyzes whether the constructed constraints match the ones in the I-block. If the verifier detects a complete match, the code in the I-block is executed. Otherwise, the verifier sends an abort signal to the processor which can be resolved in several ways: for example, non-blockable exception or system reset. In the former case, the OS on the processor can then terminate the process.

The constraint verification procedure is performed by on-chip hardware. The constraint verifier can be implemented in numerous ways. For example, the I-block buffer can be multiplexed from the set of instruction cache lines. Instructions must be verified every time a new I-cache line is loaded. Since writes to the I-cache are not possible, verification is not necessary for repeat accesses to any of the lines. This approach may have adverse effect on processor's critical path due to additional logic attached to the I-cache. Another implementation is to have the I-block buffer precede the I-cache as a pre-fetch unit. In addition, the results of several units in the processor such as the out-of-order execution unit (basically, its scheduler) can be reused in the SPEF verifier [26]. Then, the hardware that supports speculative execution can be used to compensate for the delay caused by the SPEF engine by executing the freshly loaded I-block, but not committing the actual transactions until the SPEF verifier does not validate the code [25].

For the sake of brevity and simplicity, in this paper we adopt the first implementation with a pipelined constraint verifier as the only performance improving technique. For the set of constraints that we have considered in our implementation of SPEF, we estimated that the verifier should not exceed 20K gates, thus inducing little overhead to the hardware design. However, our particular implementation of SPEF induces non-trivial delay as constraints are verified for every I-block fetched into the I-cache upon a miss. In our implementation, the pipelined constraint verifier which interleaves constraint verification with data fetching from main memory results in a 50-clock-cycle delay (detailed in Section 4.1). Typically, execution of many common applications for embedded systems have high I-cache hit rates; hence, the overhead of SPEF on actual run-time is frequently within 20% of the performance of the traditional system.

We quantify the effect of SPEF on the performance of standard multimedia tasks [13] implemented for the ARM instruction set and the ARM TDMI720 processor [1].

# 4. SPEF - AN IMPLEMENTATION

In this section, we present the technical details of an implementation of the SPEF framework for the ARM instruction set. We have chosen the ARM hardware-software development toolkit for two main reasons: simplistic RISC-type instruction set and availability of tools that support simulation of additional logic attached to the main processing unit. It is important to stress that implementation of a SPEF-like platform for another processor and instruction set, such as Intel's x86, may pose new challenges due to sophisticated super-scalar pipelined ALUs, variable length instructions, branch prediction, multi-processor support, etc.

In this section, we detail how domain ordering, TI-hashing, and constraint generation, encoding, and verification are performed for three types of transformations and the ARM instruction set. The considered constraint types are: INSTRUCTION RESCHEDULING, BASIC BLOCK REORDERING, and PERMUTATION OF REGISTER ASSIGNMENT. They satisfy the following set of requirements:

- **High degree of freedom**. Each type of constraint should provide a rich domain for code transformation, implying a large number of distinct representations for a given I-block.
- **Functional transparency**. The induced transformations must not alter program's functionality.
- **Transformation invariance**. The encoding of constraints must depend exclusively on the functionality of an I-block and processor's CPU ID. Constraint encoding must be exactly the same *before* and *after* the constraints are embedded into an I-block. Thus, the induced code transformations due to constraint encoding must not alter the result of domain ordering and TI-hashing.
- **Effective implementation**. The hardware implementation of the constraint verifier must be fast and require few silicon gates.
- **Low performance overhead**. The imposed changes due to constraint encoding should have minimal performance overhead. For example, instruction rescheduling poses little performance overhead on processors with out-of-order execution[7], however, it may have a dramatic effect on heavily pipelined architectures. In such a case, instruction rescheduling can be used with certain additional constraints or not used at all as commonly, there is plenty of transformation freedom for a given, relatively large, I-block.

The properties of the implemented protection system were tested using a benchmark test assembled from the MediaBench test suite [13]. It included the MPEG and JPEG codecs, PEGWIT, and a set of cryptographic primitives.

## 4.1 Transformation Types

In this subsection, we present the technical details behind the embedding and verification procedures for each type of code transformation (i.e. constraint type). We analyze two basic tasks performed during software installation and execution: (*i*) domain ordering including TI-hashing and (*ii*)

[7]Out-of-order execution is a crucial component of almost every modern architecture that relies on inherited instruction sets (e.g., Intel's x86), as pipeline structures of new generation processors are usually not built with the target to achieve high performance for programs optimized for the pipelines of the old architectures.

constraint generation, encoding, and validation. For each constraint type, we provide an example and a quantification of the main security parameter: **degree of freedom** (DOF). DOF aims at modeling the entropy of an I-block with respect to the considered transformations.

DEFINITION 1. *The* DEGREE OF FREEDOM *of a given I-block with respect to a given set of constraint types quantifies the number of ways the I-block can be transformed such that the functionality (semantics) of the I-block is preserved.*

The inverse of DOF equals the probability that a random I-block of instructions (potentially malicious) accidentally satisfies all required constraints considered for constraint embedding. Thus, DOF relatively accurately models the security of the scheme. For a set of constraint types $S = \{C_1, \ldots, C_n\}$ that are mutually orthogonal, i.e. transformation in one constraint domain does not affect the DOF for another constraint domain, the total DOF of an I-block $I$, $\delta(I, S)$, equals the product of DOF for each individual constraint type:

$$\delta(I, S) = \prod_{i=1}^{n} \delta(I, C_i), \tag{1}$$

or from the perspective of the corresponding entropy:

$$H(I, S) = \sum_{i=1}^{n} H(I, C_i) = -\sum_{i=1}^{n} \log_2 \left[ \frac{1}{\delta(I, C_i)} \right], \tag{2}$$

where $\delta(I, C_i)$ denotes the number of possible distinct transformations of the I-block $I$ using the transformation type $C_i$. All transformation types evaluated in the following subsections are mutually orthogonal.

### 4.1.1 $C_1$ - Instruction Reordering

Static instruction scheduling is an optimization technique that can greatly improve program performance, especially for heavily pipelined architectures that do not have out-of-order execution units. However, the targeted ARM processor, as a single-issue architecture, is minimally affected by a particular instruction order within a basic block. Thus, we adopt instruction scheduling as a transformation for constraint embedding. In this subsection, we evaluate the two key steps in SPEF for this type of transformation: domain ordering and constraint embedding.

**Domain ordering.** For this constraint type, domain ordering assigns a unique identifier to each data-flow (non-branch) instruction. Since instructions are ordered only within basic blocks, domain ordering is performed independently for each basic block[8]. First, the data-flow instructions within a basic block are sorted with preserved dependencies using the following criteria easy to implement in hardware: e.g., ($i$) instruction op-code that includes information about the number and indexing of operands and ($ii$) instruction fan-out. A fan-out of an instruction that sets register `r`, is defined as the XOR sum of op-codes of all instructions that use the value in register `r` as a first operand. If there are no such instructions, the fan-out equals `null`. For example, consider instructions `(1)` and `(2)` in Figure 4a that cannot be distinctly sorted using the first two policies. However, the fan-out of instruction `(1)` is `null` and the fan-out of instruction `(2)` is `opcode(MUL)`. This distinction is used to sort `(1)` ahead of `(2)`. In our experiments, we have not encountered a single I-block that could not be fully sorted using the above set of policies. If certain instructions cannot be sorted distinctly (assigned unique identifiers), then they are not considered in this transformation. The ordering of the domain for the instruction rescheduling transform is formally presented using the pseudo-code in Figure 5.

[8]Please see next subsection on how basic blocks boundaries are determined at run-time.

```
1  for each instruction block B
2      S1 = sort B in decreasing order of op-codes
3      Set (∀i ∈ B)ID(i) = arg{i, S1}
4      for each subset of instructions b ∈ B
       with ∀i ∈ b| arg{i, S1} = const.
5          S2 = sort b in decreasing order of fan-out
6          Set (∀i ∈ b)ID(i) = ID(i) + arg{i, S2}
```

**Figure 5: Pseudo-code for ordering the domain of the instruction rescheduling transform. Function $\arg\{i, S_x\}$ returns the index of instruction $i$ in a sorted list $S_x$. The index of the first element in the list equals zero. Instructions that cannot be distinctly sorted, have equal, smallest possible, indices. $\mathrm{ID}(i)$ is the resulting instruction identifier.**

**Constraint embedding.** Based on the random bitstream, out of all possible permutations of instructions that preserve the functionality of an I-block, the constraint embedding algorithm selects one and reschedules the instructions according to this new instruction order. The scheduling algorithm constructively builds the ordering by selecting a specific instruction from a pool of instructions that can be executed at a particular control step.

The constraint embedding is illustrated in Figure 4. A simple code segment with six instructions illustrates the degree of freedom for instruction scheduling as a transformation domain (see Figure 4a). Each of the instructions can be scheduled in more than one control step either unconditionally (denoted in Figure 4a as $\oplus$) or conditionally ($\oslash$) as shown on the right side in Figure 4a. Control step `i` can be conditionally occupied by instruction `(a)` if there exists at least one more instruction whose scheduling in control step `j` enables scheduling of instruction `(a)` in control step `i` such that no data-flow dependency between instructions is violated. For example, instruction `(4)` can be scheduled in step `3` only if both instructions `(1)` and `(2)` are scheduled in control steps `1` and `2`. Instruction `(3)` can be unconditionally scheduled in any of the given steps since it does not have any data-flow dependency with other instructions within the selected block. Even in such a small instance, the number of possible schedules in this example is 24.

For simplicity reasons, the domain ordering used in this example is simplified to the instruction numbering of their initial schedule. The instruction ordering algorithm shown in Figure 4d constructs the ordering that represents the working-copy of the code as follows. There are three instructions that can be scheduled in the first control step: `(1)`, `(2)`, and `(3)`. The first four bits of the random bitstream are `0110`. Since there are three instructions that can be scheduled at the first control step, we use the first two bits

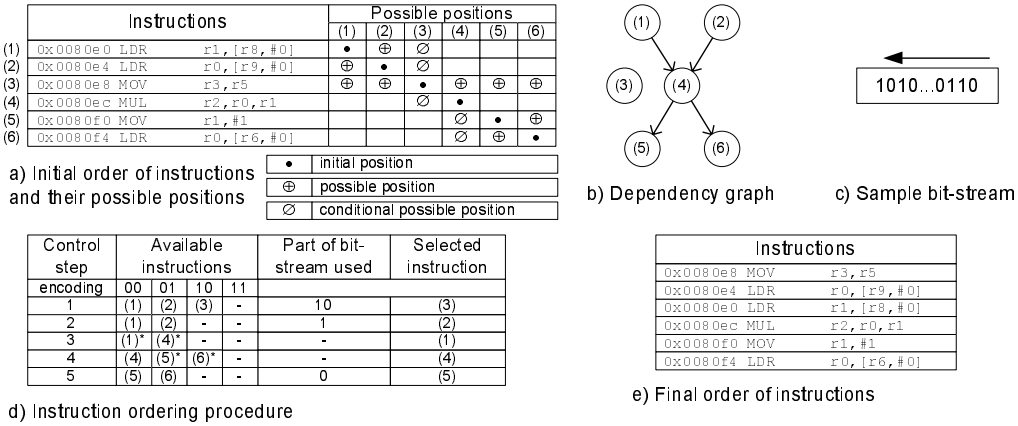


**Figure 4: An example of constraint embedding for the instruction rescheduling transform.**

from the bitstream `10` to select the corresponding instruction. In this case, according to the encoding in Figure 4d, instruction `(3)` is scheduled in the first control step. Note that the number of instructions that can be scheduled at a particular control step depends upon the preceding scheduling decisions. The process continues using the subsequent bits of the bitstream as illustrated in the example in Figure 4d. The resulting ordering is shown in Figure 4e.

**Constraint verification.** Whereas the software installer enforces the generated constraints, the constraint verifier only validates their existence. The constraint verifier performs domain ordering, generates constraints based on the extracted TI-hash, and validates that the particular ordering of instructions corresponds to the generated constraints. Domain ordering can be performed using a hardware sorter in $\log_2(n)$ clock cycles, where $n$ is the number of instructions in the I-block. From the moment the random bitstream is available, the actual verification of constraints can be performed in a single clock cycle.

Evaluating $H(I, C_1)$ is a hard task as the number of valid scheduling solutions is commonly exponentially proportional to the number of instructions within a basic block. In our experiments, we have compiled our results using a lower bound on $\delta(I, C_1)$ which is computed as follows: if the length of a basic block is shorter than 10 instructions[9], we compute $\delta(I, C_1)$ exhaustively for each block in $I$:

$$\delta(I, C_1) = \prod_{\forall block \in I} \delta(block, C_1). \quad (3)$$

Basic blocks with more than 10 instructions are partitioned into sub-blocks of up to 10 instructions. We treat each block as a basic block, and then compute $\delta(I, C_1)$ according to Eqn.3. Although this approach may grossly lower the values for $H(I, C_1)$, it still provides a good estimate on which I-blocks do not provide high $H(I, C_1)$. The results of our approximation of $H(I, C_1)$ for MediaBench applications are presented in Figure 6.

### 4.1.2 $C_2$ - Basic Block Reordering

In this subsection, we introduce basic block reordering as another transform for constraint embedding. A basic block generally refers to a section of code that has only one entry and one exit. Basic block identification is one of the elementary tasks used in optimizing compilers to improve code efficiency. In many instances, basic blocks can be relocated with little overhead. In this subsection, we describe a method to encode constraints using a specific permutation of basic blocks in an I-block.

During code verification, basic block boundaries are not known inside of an I-block unless the basic block is ended with a control transfer instruction such as a conditional branch. Thus, we make an additional requirement for program's master-copy: if a basic block does not end with a control transfer instruction, an unconditional branch to the following instruction is inserted at the end of the block. This way, the constraint verifier is able to determine the basic block boundaries necessary to extract the encoded constraints. For our benchmark applications, on the average 23% of basic blocks do not end with one of the control transfer instructions, and the size of a basic block on the average is 4.7 instructions. As a result, the insertion of unconditional branches increases the total program size 7.5%. Finally, basic blocks that span across two or more I-blocks require special attention. Such basic blocks are partitioned along I-block borders and treated as separate basic blocks.

Figure 7 shows a simple example that illustrates how basic-block reordering is used as a transformation domain. A portion of code with five basic blocks is shown. Both the initial (in Figure 7a) as well as the transformed (in Figure 7b) basic block scheduling are presented. The arrows represent a possible change of the execution order due to branching. All execution paths must exist after the reordering procedure. Thus, it is necessary to redirect some of the branches as well as change their branching condition (e.g., branch on address `0x00dce8`). In some cases, it is necessary to insert unconditional branches such as the one at the address `0x00dd20`. Reordering basic blocks results in a high degree of freedom for constraint manipulation as there exist $\delta(I, C_2) = N!$ possible orderings for an I-block with N basic blocks.

**Domain ordering.** Basic blocks in an I-block are enumerated in the increasing order of the first data-flow instruc-

[9]A block of ten mutually independent instruction can be scheduled in $10! \approx 3.6 \cdot 10^6$ different ways.

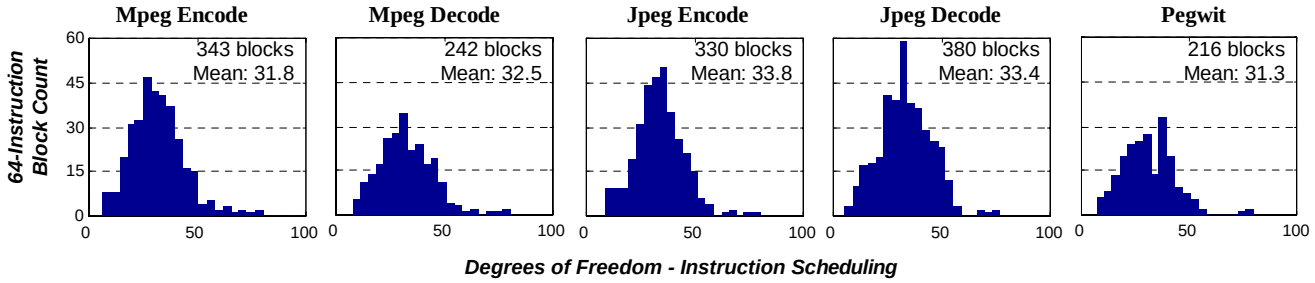


**Figure 6: Experimental results measuring $H(I, C_1)$ for instruction scheduling as a transformation type. The abscissa in each histogram represents the number of I-blocks with the DOF level specified on the ordinate.**

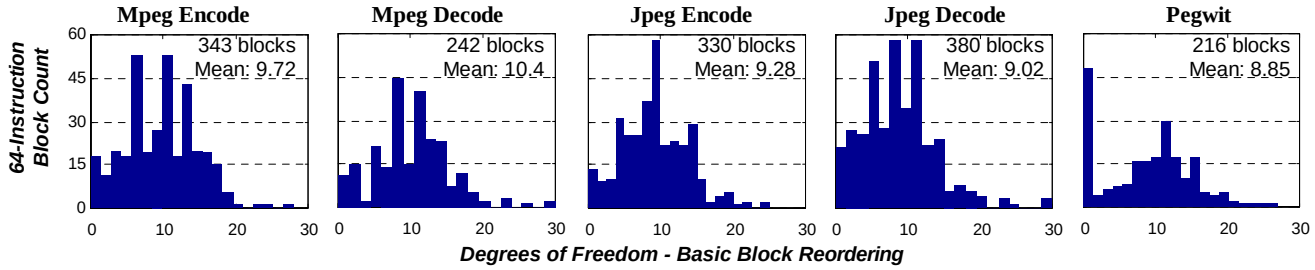


**Figure 8: Experimental results measuring $H(I, C_2)$ for basic block reordering as a transformation type. The abscissa in each histogram represents the number of I-blocks with the DOF level specified on the ordinate.**

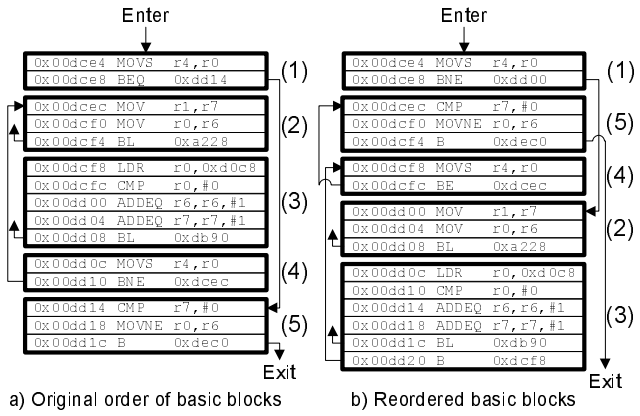


**Figure 7: An example of basic block reordering. Different basic block schedules may involve addition of unconditional branch instructions. Such case occurs in basic block (3).**

tion in each basic block. The procedure for enumerating instructions is presented in the previous subsection.

**Constraint embedding.** The bitstream determines the ordering of basic blocks using the random permutation function presented in the previous subsection. The number of bits $m$ required to select a particular permutation of $N > 2$ basic blocks using this algorithm is:

$$m = \sum_{i=2}^{N} \lceil \log_2 i \rceil < \lceil \log_2 N! \rceil. \quad (4)$$

**Constraint verification.** The constraint verifier performs domain ordering, generates constraints based on the extracted TI-hash, and validates that the particular ordering of basic blocks corresponds to the generated constraints. Domain ordering for basic block reordering waits for the enumeration result of the same procedure for instruction rescheduling. Then, it assigns a unique identifier for each basic block – this step adds $\log_2(N)$ clock cycles to the delay imposed by the domain ordering hardware for instruction rescheduling. Once the random bitstream is available, constraint validation can take potentially only a single cycle.

Figure 8 shows the experimental results measuring the parameter $\delta(I, C_2)$ for our benchmark suite of applications. In our experiments, we observed an average of 11 basic blocks per a 64-long I-block. However, in certain cases such as in the digital signal processing functions in MPEG, basic blocks can be larger than the I-block.

### 4.1.3 $C_3$ - Permuted Register Assignment

The third transform considered for constraint embedding is register reassignment. Register allocation includes assigning variables to registers during the compilation process. This problem can be modeled as graph coloring where the nodes of the graph represent variables, edges between nodes represent overlapping lifetimes between variables, and colors correspond to registers. In SPEF, during the constraint embedding process, assignment of variables which are created within a given I-block is dictated by the random bitstream. It is important to stress that this type of transformation must be performed after instruction rescheduling constraints are embedded, because the domain ordering for permuted register assignment depends upon the result of that transformation[10]. We denote the set of registers modified within

[10]Hence, the transforms need not be truly orthogonal in SPEF. The only requirement is that constraints embedded by all considered transforms must be detected properly if transforms are

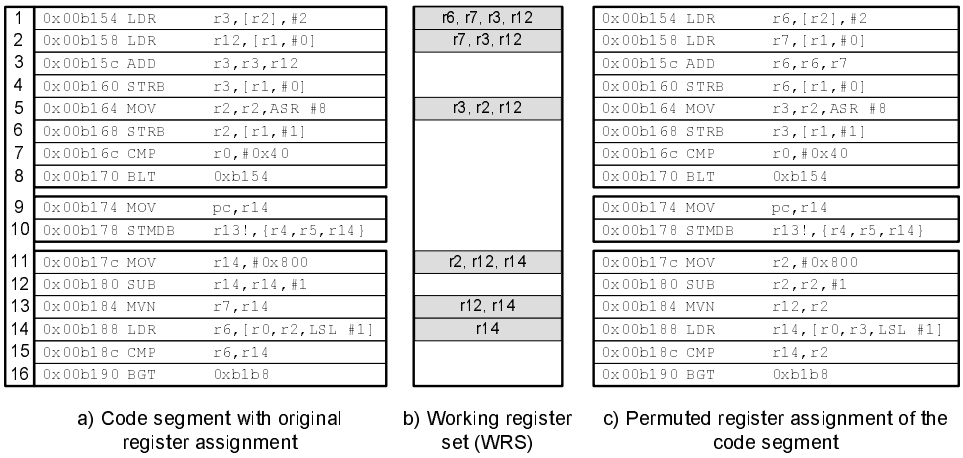


**Figure 9: Example of a block of 16 ARM instructions taken from the MPEG2 encoder. Six different registers are set in this block. At any control step, at most four registers are candidates for reassignment. During constraint verification, one out of 216 different register assignments is selected.**

an I-block as the *working register set* (WRS), which is a subset of the full register file. The working register set represents the domain for this transformation.

**Domain ordering.** A unique identifier X(i) assigned to a variable i is equal to the difference a-b where a is the address of the instruction that creates i and b is the starting address of the I-block. Registers are numbered following the ascending index in the register file.

**Constraint embedding.** Constraint embedding is performed by assigning the variables generated within the I-block, to a random permutation $\pi$(WRS) of the working set of registers. The new assignment is determined by the random bitstream, by the dependencies among variables, and by the set of available registers WRS($i$) at control step $i$. The register reassignment within one I-block must be propagated throughout all I-blocks in which the modified variables are alive. Note that the propagation is done only during constraint embedding, not their verification. Hence, it does not affect the computation of the TI-hash in the subsequent runs of the SPEF verification hardware.

We illustrate permuted register assignment using an example presented in Figure 9. A segment of ARM-code extracted from an MPEG2 encoder [13], contains 16 instructions. The original register assignment of this code segment is shown in Figure 9a. There are six distinct register assignments and the affected registers are: WRS = {r2, r3, r6, r7, r12, r14}. In the example, the constraint encoder uses the random bitstream to compute the following exemplary permutation $\pi$(WRS) = {r6, r7, r3, r2, r12, r14}. Figure 9b shows the set of registers WRS($i$) available for (re)assignment at each control step $i$. For example, in the first control step, we have: WRS(1)={r3, r6, r7, r12}. Registers r2 and r14 are not elements of WRS(1) because they are used as sources in instructions prior to their assignment in the block.

The constraint encoder parses the I-block top-down and at each control step at which a new register is assigned to a variable, it selects a register from WRS($i$) with the smallest index in $\pi$(WRS). For example, register r6 is assigned to the variable created by the first instruction in the I-block. The used register replaces all appearances of the replaced variable. Namely, r6 replaces all occurrences of r3 in the block. Register r6 is then removed from WRS. Registers that are used as source registers prior to their assignment, are added to WRS as they become available. For example, register r2 is added to WRS in control step 5. The resulting register assignment is illustrated in Figure 9c.

**Constraint verification.** Domain enumeration is a trivial task for this type of transformation as variables are sorted in the order they are generated and registers inherit their register file indices. The extraction of WRS as well as the permutation selected using the random bitstream, can be computed in a single cycle. In at most |WRS| steps, we can sequentially verify whether a particular variable has been assigned to a proper register as guided by the random bitstream. Assuming the random bitstream is available, the verification procedure can be performed in $1 + |\text{WRS}|$ cycles. As this number is limited by the cardinality of the working register file, for the ARM architecture the verification of the permuted register assignment is limited to seventeen control steps. The degree of freedom for this transformation type, $\delta(I, C_3)$, is equal to:

$$\delta(I, C_3) = \prod_{i=1}^{N} |\text{WRS}(i)| \leq N! \tag{5}$$

where $N$ is the number of modified registers within the I-block and $|\text{WRS}(i)|$ is the cardinality of the available subset of registers from the working set at control step $i$. Accordingly, for the code segment presented in Figure 9, the total number of all possible register reassignments is $4 \cdot 3 \cdot 3 \cdot 3 \cdot 2 \cdot 1 = 216$. Figure 10 shows the experimental results measuring the parameter $H(I, C_3)$ for our benchmark suite of applications.

### 4.1.4 Computing the Random Bitstream

For a given set of transformation types, the random bitstream that guides constraint encoding is generated in two steps. First, the information in the I-block that is invariant to considered transformations is concatenated into a

applied in particular order.

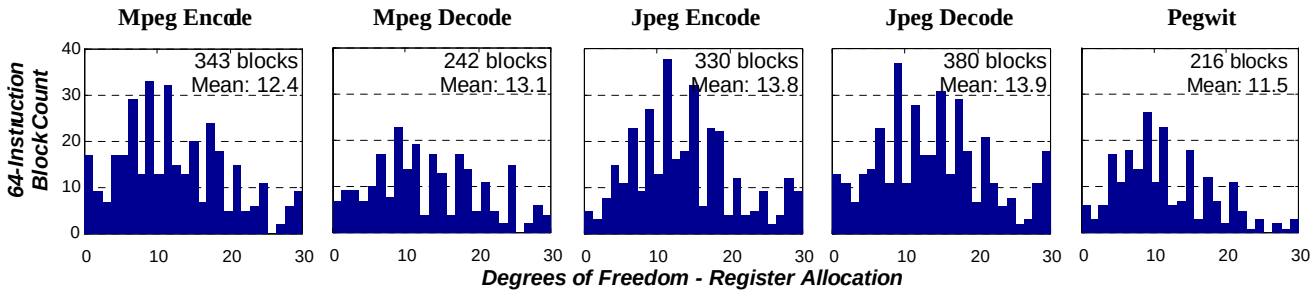


**Figure 10: Experimental results measuring $H(I, C_3)$ for register reassignment as a transformation type. The abscissa in each histogram represents the number of I-blocks with the DOF level specified on the ordinate.**

message (denoted as TI-hash). Instruction fields that contribute to this message are op-codes of data-flow instructions and constants.The message, as defined, has variable length, whereas the keyed encryption mechanism used to create the random bitstream has a fixed length input. This problem can be resolved by feeding the encryption mechanism in a CBC-MAC mode (see [15], §9.58) for as many iterations as required to process the entire message. Since each iteration takes several clock cycles (e.g., in the case of DES, a typical implementation of one iteration costs 8 cycles), this approach is not the most effective in terms of system performance. Another approach is to pipeline the encryption engine and process the message in blocks of $K$ bits where $K$ is cipher's input data width and then get the desired length of the random bitstream by XORing the encrypted blocks. In our experiments, we have observed I-blocks that demand up to 160 bits from the random bitstream to perform a full constraint encoding. Therefore, we restrict the length of the random bitstream to 256 bits.

SPEF can deploy any of the popular encryption standards which are particularly efficient for hardware implementations. Typical candidates would include a 112-bit key triple-DES or a 128-bit key AES, which create encrypted content after 8 cycles per DES iteration or 16 cycles in typical low-cost implementations respectively. Pipelined implementations would add overhead to the basic iteration period proportional to $\mathcal{O}(n)$, where $n$ is I-block length.

## 4.2 SPEF – System Security

For the sake of brevity, in this paper we do not present details of the set of additional constraint types that we have evaluated in the SPEF framework. In addition to the constraints (1-3) evaluated in Subsection 4.1, we have also studied the following additional transformation types:

4. $C_4$ – Conditional branch selection – refers to embedding constraints in an executable by selecting one of the two choices for each conditional statement that is associated to a conditional branch. For example, condition "greater-than" can be replaced by the condition "less-than-or-equal-to".

5. $C_5$ – Filling unused instruction fields with bits from the bitstream – in our implementation for the ARM instruction set, we use the software interrupt instruction (SWI) with appropriate condition codes that ensure that the instruction never causes an interrupt. Each SWI instruction contains 24 unused bits that can be set according to the random bitstream.

6. $C_6$ – Toggling the signs of immediate operands to select between addition and subtraction of in-lined constants.

The experimental results for the accumulated I-block entropy $H(I, \{C_1 \dots C_4 C_6\})$ are summarized in Figure 11. The ordinate in each histogram represents the accumulated DOF for constraint manipulation with all constraint types combined. The abscissa for each case shows the number of I-blocks with the respective DOF. The minimal and average achieved entropy across all I-blocks of all benchmark programs is 21 and 140 bits. Although the average case provides strong security, the cases with small entropy need to be addressed. A trivial transformation that achieves this goal is $C_5$. Another approach that results in significantly lower overhead, is to perform intra-I-block code perturbation in order to improve the worst case above certain lower bound (e.g., 50 bits). Algorithms that perform such optimizations are outside the scope of this paper.

The most viable attack on the system feeds the processor with an I-block which has a small entropy in terms of the considered transformations. In that case, the cardinality of the embedded constraints is small and only few bits from the random bitstream are used. This renders the authentication decision unreliable, which makes a brute force attack a possibility. To prevent this type of an attack, both the constraint encoder and verifier need to detect the case when an I-block with low entropy is encountered. This case is signalled if the sum of successfully enumerated elements in each of the domains is not sufficient (typically, less than 100). Then, the **software installer** can either: $(i)$ refuse to create the corresponding working-copy or $(ii)$ it can perturb the executable so that the constraint for minimal entropy is satisfied for all I-blocks or $(iii)$ it can improve the entropy of such I-blocks by adding dead-code which generally results in minor increase in code size and slight decrease in performance. Correspondingly, when an I-block with low-entropy is executed, the **constraint verifier** should refuse to execute this code regardless whether the constraints match.

## 4.3 SPEF – Effect on System Performance

To evaluate the impact of constraint verification on performance, we conducted a series of simulations on the ARMulator platform [1]. The system environment consisted of direct mapped and fully associative data and instruction caches of size 1, 2 and 4KB. In addition, we assumed the following system model: $(i)$ all instructions are executed in one

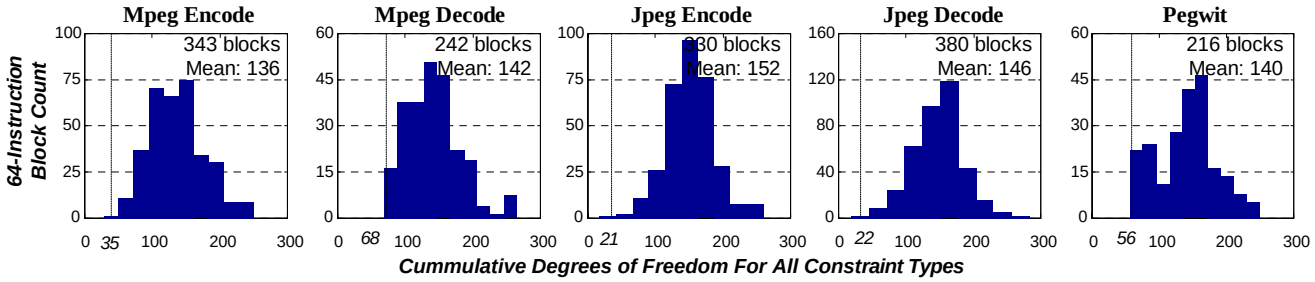


**Figure 11: Accumulated entropy $H(I,S)$ with respect to all considered transformation types – a direct measure of the likelihood that an adversary can create a malicious I-block. For each chart, the achieved minimum $\arg\min_{\forall I} H(I,S)$ is shown as a vertical dotted line.**

clock cycle if there is a cache hit, (*ii*) on a data cache miss the processor stalls $k$ clock cycles (in our examples $k = 64$ cycles for a 64-long I-block), (*iii*) on an instruction cache miss, the processor stalls $k + l$ cycles where $l$ is the component representing the delay of the constraint verifier. This models the fact that when there is an instruction cache miss, a new I-block is fetched from memory and then validated using SPEF's constraint verifier. Overhead $l$ has four components: delay due to domain ordering, TI-hash extraction, random bitstream generation, and verification. Discussion on their hardware implementation can be found in Subsection 4.1, however, detailed descriptions of these components are beyond the scope of this paper. In our implementation, domain ordering and TI-hash extraction consume 33 clock cycles in the worst case. Random bitstream generation takes 16 clock cycles [21] and verification consumes 1 clock cycle. We used the Mediabench suite of benchmark programs as payload to the system [13]. Cache performance results for our benchmark applications under different system configurations are shown in Figure 12.

Augmenting the normal execution cycle with constraint verification increases system's clock cycles per instruction (CPI) when there is a cache miss. By adding a TI-hash cache, we have been able to reduce the penalty of domain ordering and TI-hash extraction. In our implementation, the number of entries in the TI-hash cache is set to twice the number of entries in the I-cache. That is acceptable since TI-hash entries are much smaller (256-bit payload). As a consequence, in our experiments only 20% of I-cache misses resulted in the recalculation of the domain order and TI-hash. Therefore, the performance overhead was reduced from the range 12.7%-24.7% without, to range 7.5%-17.1% with the TI-hash cache. The experimental results for different cache sizes and organization are shown in Figure 13.

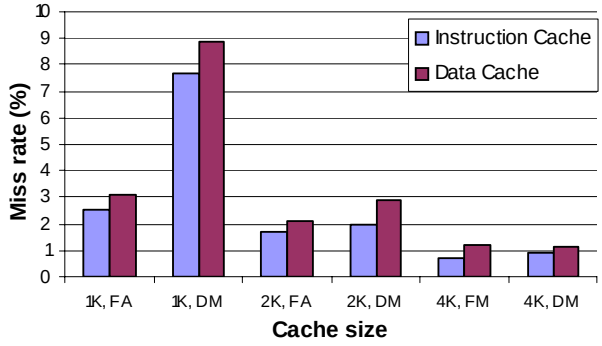


**Figure 12: Cache miss rate for different sizes of caches with direct mapped and fully associative organization.**

## 5. RELATED WORK

In this section, we review the related work for code security. Four major approaches for code security have emerged: code signing, sandboxes, firewalling, and proof-carrying code. Code signing is conceptually the simplest mobile code security technique. Code signing follows a typical authenticated handshake protocol such as WTLS [20]. Recently, sandboxing, as a security paradigm, has attracted a great deal of attention, mainly due to its applicability in Java. Although Java provides a new concept of a protected domain and several related security primitives, the backbone of its security architecture is a sandbox [23]. Sekar and Uppuluri developed a security layer that includes a sandbox designed to protect the application against malicious users and the host from malicious applications [23]. The main idea behind the firewalling technique for mobile code security is to conduct comprehensive examination of the provided mobile code at the very point where it enters the consumer domain. Several variants of the generic approach have emerged, but their effectiveness is not often satisfactory [14]. Necula [17] has developed the concept of proof-carrying code, a mechanism by which a host system can determine with certainty that it is safe to execute a program provided by an un-trusted source. This is accomplished by requesting that the source provides a safety proof that attests to the code's adherence to a consumer defined safety policy.

## 6. CONCLUSION

We have developed a system of hardware-software mechanisms that prevents execution of unauthorized code in protected (trusted) mode by imposing a computationally intractable task to a potential attacker. The key protection mechanism is space and time efficient checking of blocks of executable code with respect to a set of constraints that facilitate rapid detection of malicious code alternations. Efficient code verification is conducted at run-time by embedding the constraints into code through local syntactic changes. The installed software is individualized for each processor and the key to individualization is stored as a

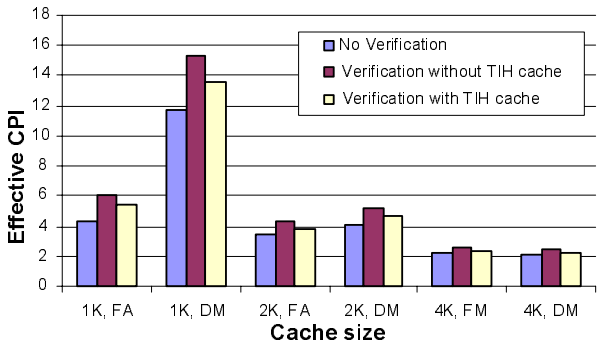


**Figure 13: Effective clock cycles per instruction (CPI) with and without the TI hash (TIH) cache.**

secret on the protected system. The protection system is transparent with respect to the OS and the applications. The changes to the executable due to individualization are performed as a post-compilation step and fully preserve the functionality of the program. The approach is prototyped and evaluated using the ARM instruction set, a corresponding embedded system, and the Mediabench set of embedded benchmark programs.

## 7. ACKNOWLEDGEMENTS

We would like to gratefully acknowledge the efforts by Seapahn Megerian in helping us obtain the experimental results. We thank Jim Larus and the anonymous reviewers for discussions that improved the content of the manuscript.

## 8. REFERENCES

[1] ARM Corp. The ARM hardware-software development kit. Available online at `http://www.arm.com`.

[2] N. Borisov, I. Goldberg, and D. Wagner. Intercepting mobile communications: the insecurity of 802.11. *MOBICOM*, 2001.

[3] S. Chari and P.-C. Cheng. Bluebox: A policy driven, host-based intrusion detection system. *Network and Distributed System Security*, February 2002.

[4] H. Chen, D. Wagner, and D. Dean. Setuid demystified. *USENIX Security Symposium*, 2002.

[5] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, and P. W. Q. Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. *USENIX Security Symposium*, pages 63–77, Jan. 1998.

[6] C. Cowan, F. Wagle, P. Calton, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. *DARPA Information Survivability Conference and Exposition. IEEE Computer Soc*, 2:95–107, 2000.

[7] D. Evans. Static detection of dynamic memory errors. *Programming Language Design and Implementation*, pages 44–53, 1996.

[8] D. Evans, J. Guttag, J. Horning, and Y. Tan. LCLint: A tool for using specifications to check code. *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 87–96, 1994.

[9] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications. *USENIX Security Symposium*, pages 1–13, July 1996.

[10] Intel Corp. *Processor Serial Number Technical Notes*. Available on-line at `http://www.intel.com`.

[11] S. Johnson. Lint, a C program checker. Unix Programmer's Manual, AT&T Bell Laboratories, 1978.

[12] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. *USENIX Security Symposium*, pages 177–89, Aug. 2001.

[13] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. *International Symposium on Microarchitecture*, 330–351, 1997.

[14] D. Martin, Jr, S. Rajagopalan, and A. Rubin. Blocking java applets at the firewall. *Network and Distributed System Security*, pages 16–26, 1997.

[15] A. Menezes, P. V. Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, FL, October 1996.

[16] R. Minnich. *The Linux BIOS Home Page*. Available on-line at `http://www.acl.lanl.gov/linuxbios`.

[17] G. Necula. Proof-carrying code. *Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[18] A. One. Smashing the stack for fun and profit. *Phrack*, 49, 1996.

[19] Phoenix Technologies Ltd. *System BIOS for IBM PCs, Compatibles, and EISA Computers*. Addison-Wesley, Reading, MA, 1991.

[20] A. Rubin and D. Geer, Jr. Mobile code security. *IEEE Internet Computing*, 2(6):30–34, 1998.

[21] Sci-Worx GmbH. AES Rijndael core. Available on-line at `http://www.sci-worx.com`.

[22] D. Seeley. The internet worm, password cracking: a game of wits. *Communications of the ACM*, 32(6):700–3, June 1989.

[23] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. *USENIX Security Symposium*, pages 63–78, 1999.

[24] U. Shankar, K. Talwar, J. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. pages 201–20, 2001.

[25] M. Smith. Support for speculative execution in high-performance processors. *PhD thesis, Stanford University*, 1992.

[26] R. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal*, pages 25–33, 1967.

[27] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. *Network and Distributed System Security*, 2000.

[28] C. Wilson and L. Osterweil. Omega - a data flow analysis tool for the C programming language. *IEEE Trans. on Software Engineering*, 11(9):832–8, 1985.

[29] Zero Knowledge Systems Inc. *The Intel Pentium III Exploit Page*. Available on-line at `http://www.zeroknowledge.com/p3/home.asp`.