A Hardware-Software Platform for Intrusion Prevention

Milenko Drinić and Darko Kirovski
Microsoft Research
{mdrinic, darkok}@microsoft.com

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, its execution flow can be redirected to externally injected malicious code using, for example, a buffer overflow exploit.

We introduce a novel, simplified, hardware-assisted intrusion prevention platform. Our platform introduces overlapping of program execution and MAC verification. It partitions a program binary into blocks of instructions. Each block is signed using a keyed MAC that is attached as a footer to the block. When the control flow reaches a particular block, its instructions are speculatively executed, while dedicated hardware verifies the attached MAC at run-time. The computation state is preserved during speculative execution using a mediating buffer placed between the processor and L1 data cache. Upon MAC verification, the results from this buffer are propagated externally. Central to this paper is the proposal of a novel optimization technique that initially identifies instructions that are likely to stall execution, and reorders basic blocks within a given instruction block to minimize the execution overhead. While the presented optimization technique is problem specific, it is flexible such that it can be adjusted for different optimization goals. Preliminary results showed that our optimization methods produced an average overhead reduction of 60% on the SPEC2000 benchmark suite and Microsoft Visual FoxPro.

1. Introduction

The key problem to security of modern computing 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 [20]. Once the adversary executes injected code in the highest trust mode, usually all system resources are at her disposal. Ease of implementation and effectiveness have established attacks that focus on redirecting program execution as the most common threat to system security [19, 5]. Aside from intrusion prevention, 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 potentially 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 [16], or perform static analysis on binaries to verify that they do not have buffer overflow exploits [10].

1.1. Intrusion Prevention - Previous Attempt

Intrusion prevention systems aim at forcing the adversary to solve a difficult problem, preferably intractable, in order to be able to run a program with desired functionality on the target computer.

Recently, Kirovski et al. introduced a computing platform that enables intrusion prevention [8]. Their platform, SPEF, uses a framework of architectural and compilation mechanisms to ensure software integrity at run-time. During software installation, SPEF computes a keyed message authentication code (MAC) [1] for every block of instructions. The MAC is encoded within the associated block of instructions. The MAC is keyed with a key unique for a given processor. The key is burnt-in and not accessible to any application except the software installer. The software installer operates exclusively in a single-process mode which cannot be accessed or interrupted by any other system or user process. At run-time, each block is verified for integrity by decoding its MAC and verifying that the existing block corresponds to the originally signed data.

The platform proposed by Kirovski et al. has three important disadvantages. First, its MAC verification mechanism is complex. Second, due to block preprocessing and MAC decoding and verification, their system imposes a clock cycle penalty approximately three times the cycle count required to verify only the MAC. Finally, the verification hardware is on most processors' critical paths, e.g., it

Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37 2004)
1072-4451/04 $20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

intertwines with the instruction buffer, scheduler, and possibly with the L1 I$; hence, it can cause relatively high performance overhead on super-scalar and pipelined machines.

1.2. A Simplified and More Effective Platform

In this paper, we propose a new intrusion prevention system that retains the security aspects of SPEF, while significantly improving upon its performance overhead. The basic components of the new system are illustrated in Figure 1. The system security mechanism is similar to the one proposed with SPEF. In order to run an executable in one of the protected modes, a privileged user of the system must install the executable. During software installation, the executable is partitioned into atomic execution units – instruction blocks (I-blocks). For each I-block, the installer computes a keyed MAC [1] and attaches it as a footer to the I-block. The size of the I-block is such that when the MAC is attached to it, its total length does not exceed the size of a single line of the L1 I$. As opposed to SPEF, the new system does not reduce the code size overhead by storing bits of the MAC in the I-block [8].

Once the protected binary is created, it can run in one or more protected modes allowed by the operating system (OS) for a given user (Figure 1). The protected binary is executed by following the normal control flow. The integrity of executed instructions is verified on-the-fly for each I-block freshly uploaded into the L1 I$. A single I-block is considered an atomic execution unit because even in the case when only a single instruction is executed from a given I-block, the result of this instruction is not propagated to the computation state that resides outside of the processor until the entire I-block is verified. After the I-block is verified, the results of speculatively executed instructions are committed. In case the integrity check fails, the processor aborts the current process. Details on the adopted computation model can be found in Subsection 2.2.

The OS uses our platform to ensure that users are not able to switch either from their assigned user mode into another, higher security level mode or to impersonate other users. The developed system represents a fundament for secure execution of programs as it can police the crucial two trusted modes on a given platform: *trusted OS kernel mode* which cannot be accessed by any user and all variants of *user modes* (private and shared). Software that is not protected, can still run on the computing machine but not in the protected modes. It can only run in a distrusted *public mode*. OS modes are described in further detail in [8], Subsection §1.2.

In this paper, we also focus on techniques that reduce the performance overhead of the above system. Software optimization in our platform deploys two main mechanisms to improve performance of software executed in protected modes. The first mechanism is hardware-assisted speculative execution (Figure 2). When an L1 I$ miss occurs, a new I$ line is loaded and while the verification hardware processes the new entry, the processor executes the freshly loaded instructions. In case of a write to external memory, if the current I-block is not verified yet, the processor stores the value and address into a mediating FIFO buffer. If this buffer is full, the processor stalls until the current I-block is verified. Upon verification, all values from the mediating buffer are propagated into the external memory space. An additional constraint is that control flow from one I-block cannot be transferred onto another until the verification of the current I-block is completed. Consequently, a processor may stall its execution due to a full mediating buffer or due to an early exit of the control flow from the current I-block.

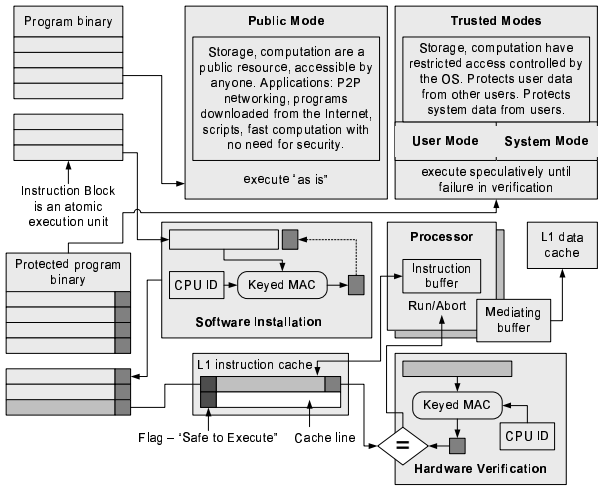

Figure 1. Main components of the proposed intrusion prevention framework.

The second optimization mechanism is a post-compilation step. Using a novel profiling methodology, we build an execution model that guides the relocation of basic blocks within a binary so that performance of the program within the new architecture is optimized (Section 3).

The adversary can operate either remotely or locally. In the first case, she scans the remote computer's ports for networking services with known security flaws and then penetrates the system using these flaws. In the second case, the adversary is already running a program on the remote system, but not in the desired mode. Thus, she 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. The most common type of such attacks are buffer overflows. For example, the simplest buffer overflow attack, stack smashing [20, 19], overwrites a buffer on the stack to replace the function call return address with the address of the injected malicious procedure. We refer the reader to [8], Section §3 for further details on the attack model.

It is important to stress that our system does not prevent nor detect buffer overruns. Thus, techniques with such a

IEEE COMPUTER SOCIETY

goal (e.g., StackGuard [5]) can be used in conjunction with our platform. Our platform prevents the adversary from executing a single line of her code in a protected mode. Our platform forces the adversary to jump into binaries that are already running in protected mode and feed them with desired data to perform malicious actions. By (re)loading programs at random locations in operating memory, this task can be made difficult.

2. System Details

In this section, we review and discuss the basic components and assumptions of the our system.

Software delivery. Two essential procedures associated with software delivery are: initial installation and updates. An application is distributed to users as a compiled binary - we denote it as the *master-copy*. A recipient of the master-copy validates its authenticity via standard public-key authentication methods [1] or proof-carrying codes [18]. The master-copy can run in a protected mode only after it is installed. We denote this copy as the *working-copy*, which is functionally equivalent to the master but augmented with processor-specific MACs for each I-block. The same procedure is performed for protected software updates.

Processor-unique secret key - CPU ID. The root of system security is a read-only register with a secret key that is unique for each processor. Kirovski et al. discuss the non-impact of CPU ID on user privacy ([8], Section §3.1).

Software installation. Software installation consists of several steps illustrated in Figure 1. First, the processor enters a special *installation mode* (i-mode). In i-mode the processor augments a given master-copy with processor-unique MACs, stores the resulting working-copy in system's memory, and finally, exits i-mode to return control to the caller, the OS kernel. The CPU ID, as a system's master secret, must never leave processor's pins. Since the installer must access the CPU ID to create the working-copy, i-mode ensures that the CPU ID is kept secret from soft and most hardware attacks. This is done by following a simple recipe: (*a*) installation is executed atomically, i.e., without any interrupts, (*b*) the installer must not write the CPU ID or any other variable that discloses plain-text bits of the CPU ID off chip, and (*c*) before completion, the installer must overwrite all intermediate results or variables stored in on-chip memory. An example of steps that a processor should follow to enter and exit i-mode is given in [8], Section §3.2.

2.1. Cryptographic Primitives

Our platform has only one cryptographic primitive, a keyed message authentication code (MAC). The purpose of a MAC is to facilitate, without the use of any additional mechanisms, assurances regarding both the source of a message and its integrity [1]. Our platform does not require public-key signatures [1] because it operates under the assumption that the CPU ID is never disclosed to the external world. We assume that the MAC is robust with respect to known-text attacks [1].

Secure MACs can be built by pipelining an n-bit block cipher in a CBC-MAC mode (see [1], §9.58). The input string is partitioned into n-bit blocks, each block is XORed with the output from the previous stage and fed as input to the current stage in the pipeline. For this purpose, we adopt a 128-bit Rijndael cipher. An exemplary ASIC implementation of this cipher loads into its pipeline a 128 bit word every cycle with an initial latency of 14 cycles [2]. Assuming an eight byte L1 I$ to L2 $ bus, we adopt the following latency model for our verification hardware:

$$t_D = 14 + \frac{\text{L1 I\$ line size}}{128 \text{ bits}} \text{ cycles.} \quad (1)$$

For a 256B L1 I$ line, the expected delay t_D is 30 cycles. Not all of the resulting 128 bits of the MAC need to be used. At the cost of reduced, however, still strong security, our platform may chose to use a subset of the available 128-bits. Security implications of such a decision are beyond the scope of this paper.

Attaching the MAC to an I-block. MACs can be attached to I-blocks in several ways: (*i*) as a footer to the I-block and insert an instruction that circumvent the control flow around the MAC; (*ii*) create a separated associated file with I-blocks' MACs which is accessed during run-time verification using a secure co-processor; and (*iii*) to have hardware support in the form of an automatic update of the program counter once it hits the last viable instruction in an I-block. In the last case, the control flow would be automatically transferred to the first instruction in the I-block located consecutively in the virtual memory. For conceptual simplicity, we assume the third alternative as a basic control flow mechanism.

Key management. In order to enable user protected modes, a distinct secret key must be used for every individual user. In addition, a given program intended to run in a protected mode must be installed for all users individually. All restrictions related to secrecy and privacy for CPU ID management discussed earlier in this section also apply to user keys. Hence, we assume that the processor is using an external non-volatile memory to store the user keys. When a user is executing a program in protected mode, the processor is responsible to load the appropriate key from the key-store in order to verify the binary. User versions of the same binary are instrumented on-the-fly before program execution from a file which contains MACs for each (I-block, user) pairing. This action is done by the OS in the highest access mode. For brevity, we present results as if only one key, the CPU ID, is used in the system.

2.2. Computation Model

In this subsection, we review the computation model used to demonstrate how our platform works and quantify

IEEE COMPUTER SOCIETY

its performance characteristics. Basic components of the computation model are presented in Figure 2. Since the atomic execution unit is an I-block, we model all relevant events related to fetching and executing an I-block.

First, we define an *I-block* as a sequence of instructions the length equal to the system's L1 I$ minus MAC's length. One I-block contains at least a part of one basic block. General expectation is to have several basic blocks contained by an I-block. One basic block can span across one or more I-blocks.

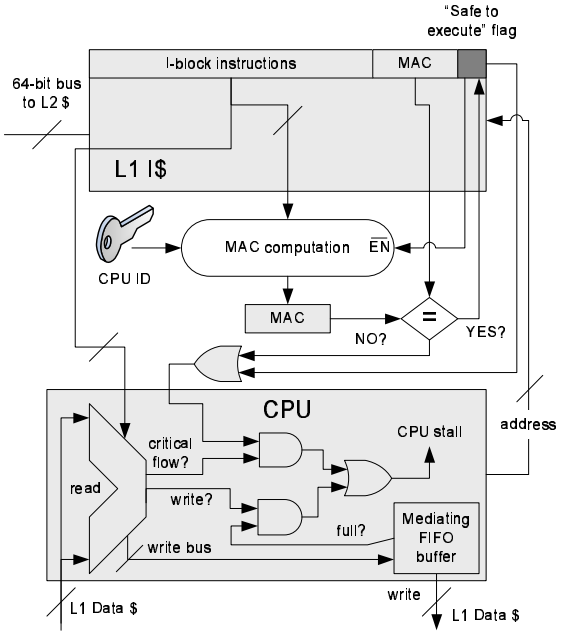

Figure 2. The computation model encompasses the L1 I$, the processor, and the cryptographic unit.

We adopt the single clock cycle per instruction (1CPI) model which assumes that one instruction is issued, fetched, executed and its result sent to a memory write buffer in a single cycle. Although this model is restrictively simple, it captures well the performance of modern pipelined and super-scalar processors such as the Pentium IV [4]. The model does not account for the delay required to load processor's pipelines after an L1 I$ miss is resolved. Since both platforms with and without MAC verification experience this additional overhead, realistic performance overhead is upper bounded by the 1CPI model. The design tendency past several decades has been to increase the number of issued instructions per clock cycle, with recent processors experiencing strong sub-1CPI performance. A fixed delay t_D due to MAC verification would affect these systems more dramatically. The expectation is that more gates available for the MAC verifier would speed up its operation as well.

Our second assumption is that the size of the L2 $ is infinite and that the entire program is loaded into the L2 $ prior to its execution. Just as with the 1CPI computation model, this assumption puts an upper bound on the performance overhead. Delay due to L1 $ misses is modeled as follows:

$$t_{\$} = 4\frac{\text{L1 \$ size}}{128 \text{ bits}} \text{ cycles}, \quad (2)$$

where we assume an individual 128-bit bus between both the data and instruction L1 $ and the L2 $. We also assume that this bus is operating at a clock rate four times slower than the processor clock.

load instruction I from memory address m indexed
by processor's program counter (PC)
if data from m is not cached in L1 I$
wait $t_{\$}$ due to I$ miss
verify MAC(**l**) of new I$ line **l**
if I is critical according to criteria (ii–iii) ♦
if MAC(**l**) is not verified
wait until MAC(**l**) verification done
if MAC(**l**) is invalid **then** abort process
execute I
if I produces a memory write
if MAC(**l**) is not yet verified
if mediating buffer is full
wait until MAC(**l**) verification done
if MAC(**l**) is invalid **then** abort process
else write data, mem-address to mediating buffer
else write data to L1 D$
if MAC(**l**) verified, mediating buffer non-empty, D$ idle
write data from mediating buffer to L1 D$
increment PC with smart skip of MAC(**l**)

Table 1. Control flow in a processor enhanced with our platform.

A processor enhanced with our platform executes instructions following an algorithm described using the pseudo-code in Table 1. According to the 1CPI computation model, the flow in Table 1 is assumed to last one clock cycle unless the processor ends up in one of the *wait* states. In addition, I$ lines are verified only after being fetched into the I$. A single "safe to execute" bit attached to a cache entry, is sufficient to denote this information back to the processor.

The computation flow also refers to critical instructions (referral denoted as ♦). A *critical instruction* as an instruction that can cause a processor to stall. Such instructions include:

(i) all memory writes,

(ii) all conditional branches and jumps onto non-cached addresses, and

(iii) the last instruction in an I-block that precedes the MAC.

Performance improvement techniques such as prefetching, out-of-order execution, etc., may impose adverse effects on our platform. For brevity and conceptual simplicity we disregard this issue which should not be neglected with all considerations for practical implementation.

IEEE COMPUTER SOCIETY

SPEC	Exec. time [sec]	Basic Block Level Profile									
		Total Blocks	Live		Significant		W/ Write [%]	Branch Taken [%]	λ/α [%]		$\lvert\frac{\lambda}{\alpha} - \frac{1}{2}\rvert < 0.2$ [%]
			count	[%]	count	[%]			> 0.3	< 0.1	
gzip	177	4832	1633	33.8	951	19.7	33.5	34.6	9.3	73.8	8.4
vpr	231	10495	2052	19.6	502	4.8	34.1	13.8	5.3	94.1	4.1
mcf	263	3971	1411	35.5	583	14.7	28.7	18.6	2.8	97.2	1.4
perlbmk	167	43075	15928	37.0	11498	26.7	36.0	12.2	3.4	88.9	3.3
vortex	159	25513	12217	47.9	7506	29.4	25.3	25.2	0.7	97.3	0.6
bzip2	209	4449	1793	40.3	1009	22.7	30.9	36.0	2.9	94.3	2.9
twolf	425	14671	5765	39.3	3669	25.0	39.1	32.7	2.2	95.7	1.6
FoxPro	6	363937	30216	8.3	2203	0.6	25.3	27.8	6.2	82.4	5.0

Table 2. Basic block and instruction level profile of SPEC2000 benchmarks and Microsoft Visual FoxPro. The columns display the total number of basic blocks in a binary, actually executed blocks("Live"), blocks executed at least 100 times ("Significant"), percentages of blocks with at least one branch instruction and taken branches. Last three columns represent the effect of λ parameter: the ratio of number of branch toggles (λ) vs. total number of executed blocks (α) greater than 0.3, smaller than 0.1, and with $\lvert\frac{\lambda}{\alpha} - \frac{1}{2}\rvert$ smaller than 0.2 respectively.

2.3. Program Profiling

In order to position the optimization goals, we adopt a program profiling method which models relatively accurately the optimization problem. We use the SPEC2000 benchmark applications and Microsoft Visual FoxPro compiled for the x86 instruction set in order to demonstrate the effectiveness of our platform. Instructions in this set are of variable length, hence, we use rescheduling and NOP padding to ensure that no instruction is spread over two I-blocks. The profile is computed dynamically using data-sets provided with our benchmarks. The profile information is collected for each basic block b_i and encompasses:

$\alpha(b_i)$ – number of times b_i executed,

$\beta(b_i)$ – number of times a branch at the end of b_i was resolved as true,

$\gamma(b_i)$ – a flag denoting whether b_i is a self-looping block,

$\delta(b_i)$ – index of the first critical instruction in b_i, assuming its critical nature is due to criterion (i),

$\lambda(b_i)$ – if b_i ends with a conditional jump, we record the number of times the decision reached by that branch toggled in the subsequent run,

$\chi(b_i)$ – we record the average block self-execution distance (*bsed*): a number of cycles passed until the same block executes again.

The complexity of collecting this profile per block is $\mathcal{O}(1)$ by using separate counters for α, β, and λ. Parameters γ, δ, and λ require an additional flag. Parameter χ requires two registers, one to store the cycle count of its last execution and another to store the sum of all *bsed* occurrences during program's execution. While the $\{\alpha, \ldots, \lambda\}$-profile is a standard way of modelling program execution, to the best of our knowledge, we are the first to propose basic block relocation algorithms based on block self-execution distance.

3. Software Optimization for Intrusion Prevention Platform

In this manuscript, we focus on software optimization techniques that aim at remedying the performance overhead due to run-time MAC verification. The main optimization technique is centered on relocating basic blocks so that once a new I$ line is loaded, the control flow within this line reaches a critical instruction of type (ii-iii) after at least t_D clock cycles. Another goal is to minimize the number of speculatively executed writes to memory before I-block's MAC is computed. Since relatively small mediating buffers managed not to overflow on memory writes in most applications which we executed on our platform, we scaled appropriately the importance of this optimization goal.

First, we formally define our optimization goal. For a given set B of N basic blocks $B = \{b_1, \ldots, b_N\}$, we seek for their permutation $\pi =< b_1^\pi \ldots b_N^\pi >$ which minimizes the following delay metric:

$$\arg\min_{\pi} \sum_{i=1}^{M} \frac{\phi(I_i)}{\alpha(I_i)} \sum_{j=1}^{|P|} \alpha(I_i, p_j) \cdot \mathrm{limit}\left[t_D - \delta(I_i, p_j)\right], \qquad (3)$$

$$\mathrm{limit}(x) = \begin{cases} 0 & , \quad x < 0 \\ x & , \quad \text{otherwise} \end{cases},$$

where $\phi(I_i)$ and $\alpha(I_i)$ are the total number of cache misses and executions respectively for I-block I_i, M is the total number of I-blocks in tile π, and I_i is an individual I-block from the tiling. We define a set of *entry points* of an I-block as a subset of all basic blocks within an I-block that have preceding points in the control flow graph outside of the I-block. If an I-block contains only a part of a basic block, this part is treated within this I-block as a separate basic block. We further denote as P the set of all feasible *critical paths* $P = \{p_1, \ldots, p_{|P|}\}$ within the internal control flow graph of a given I-block. A critical path is a path in the control flow graph of an I-block such that there exists exactly one basic block in this path which contains at least

IEEE COMPUTER SOCIETY

one critical operation. The first basic block in a critical path must belong to the set of entry points. Parameter $\alpha(I_i, p_j)$ in Eqn. 3 denotes the number of times path $p_j \in I_i$ executed in the profile. Parameter $\delta(I_i, p_j)$ quantifies the timing in cycles before the first critical instruction in $p_j \in I_i$ is reached.

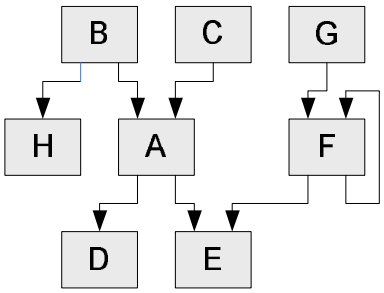

Figure 3. An example control flow graph at the basic block granularity.

3.1. Optimization Trade-offs

In order to address the optimization goal, we first review the most important trade-offs involved in making relocation decisions that affect performance at our platform. The main trade-offs are discussed using an example control flow graph in Figure 3. First, we introduce a *constraint* $\rho(b_i)$ *of a basic block* b_i as a heuristic quantifier that measures the performance overhead that b_i can cause if relocated randomly. We evaluate the dependency of ρ with respect to the collected profile and control flow graph. Intuitively, frequently executed basic blocks ($\alpha \uparrow$) as well as basic blocks terminated with a branch that cannot be predicted accurately ($|\beta/\alpha - 1/2| \downarrow$ and $\lambda \uparrow$) should be considered of high constraint. In addition, self-looping blocks ($\gamma = 1$) which do not contain a critical instruction ($\delta = 0$) should have lower constraint because they cause large number of instructions to be executed before any critical operation is issued. Basic blocks which contain a critical instruction of type (i) early on ($\delta \downarrow$) are treated as constrained blocks. Basic blocks with low *bsed* ($\chi \downarrow$) execute often in bursts, hence, should be relocated with more attention. In order to quantify ρ for a basic block, we have empirically derived the following metric:

$$\rho = \frac{(-1)^{f(\delta,\gamma)}\alpha}{\left(C_\lambda + \left|\frac{\lambda}{\alpha} - \frac{1}{2}\right|\right)\left(C_\beta + \left|\frac{\beta}{\alpha} - \frac{1}{2}\right|\right)\left(C_\chi + \frac{\chi}{\bar{\chi}}\right)}, \quad (4)$$

where constants C_β, C_λ, and C_χ all equal one, function $f(\delta, \gamma)$ returns 1 if $\delta = 0$ and $\gamma = 1$ and zero otherwise, and $\bar{\chi}$ denotes the average χ computed over the subset of all basic blocks B which are not self-looping and which are executed at least $0.1 \cdot \max_B(\alpha)$ times. For basic blocks executed less than $0.1 \cdot \max_B(\alpha)$ times, $\chi/\bar{\chi}$ is set to zero.

Different trade-offs for basic block relocation can be extracted from the control flow graph (CFG). Consider the CFG illustrated in Figure 3. Nodes denote basic blocks and edges denote potential control flow between nodes.

FFF – frequent fan-in and fan-out concatenation. Block A has blocks D and E as its fan-out, and it has blocks B and C as a fan-in. Assuming A has a low $\lambda(A)/\alpha(A)$ with most of the control being swayed towards D, then there is an intuitive demand to concatenate A and D into a common I-block. Similarly, for $\beta(B \to A) \gg \beta(C \to A)$ we would want to concatenate B and A. However, notice that in case the other fan-out branch of B is $\beta(B \to A) \ll \beta(B \to H)$, then according to the same heuristics, it is better to concatenate B and H as opposed to A.

CAC – cache anti-collusion. If $B \to A \to D$ is a common route in the program profile and the three blocks cannot fit one I$ line, then on the average, program performance is not affected if the three blocks are stored in two different I-blocks in memory that map to non-colluding cache lines. This heuristic follows the results obtained on program hot subpaths [17].

TB – toggling branches. In the case when the control flow after block A relatively equiprobably ($\beta(A) \approx \alpha(A)/2$) and unpredictively ($\lambda(A) \approx \alpha(A)/2$) branches out to D and E, it is beneficial to store all three blocks in a common I-block or if this is not possible due to block size, into two non-conflicting I-blocks.

CSLB – constrained self-looped blocks. A self-looped block F with memory writes is particularly sensitive to relocation because of the high likelihood that the mediating buffer overflows when F is an entry point to an I-block. We place non-constrained blocks preceding F, such as G, into the same I-block to mask the verification delay t_D with instructions that can be speculatively executed at no delay cost.

The trade-offs for early avoidance of critical instructions mash with the challenge of relocating basic blocks for cache miss reduction, a problem which is well studied [14, 3, 13]. We address the two problems jointly, from both the problem definition and solution perspective.

3.2. Optimization Problem Definition

INSTANCE: Given a sequence of instructions broken into N basic blocks $B = \{b_1, \ldots, b_N\}$, program CFG and profile $b_i : \{\alpha, \ldots, \chi\}$, I-block size, $T \in \mathbb{R}$.

QUESTION: Is there a reordering of basic blocks $\pi = \langle b_1^\pi \ldots b_N^\pi \rangle$ such that partitioning of the new sequence into M I-blocks yields limited cost:

$$T \leq \sum_{i=1}^{M} \frac{H(I_i)}{LE_i} \sum_{j=1}^{E_i} \sum_{k=1}^{L} \alpha(I_i, p_j^k) \cdot \operatorname{limit}\left[t_D - \delta(I_i, p_j^k)\right], \quad (5)$$

$$H(I_i) = -\sum_{i=1}^{w} q(I_w) \log_2(q(I_w)), \quad (6)$$

$$q(I_w) = \sum_{i=1}^{v} \alpha(b_v), \quad (7)$$

1072-4451/04 $20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

where M is the total number of I-blocks that tile π and I_i represents an individual I-block from this tiling. Set of basic blocks E_i represents the entry points for block I_i. We further denote as $P_{i,j}$ the set of L most likely critical paths $P_{i,j} = \{p_j^1, \ldots, p_j^L\}$ of I_i. Usually, $L \in \{2, 5\}$. Set of paths $P_{i,j}$ is computed based on the propagation of the α-metric within the program control flow graph. Parameter $\alpha(I_i, p_j^k)$ in Eqn. 5 denotes the *expected* number of times path p_j^k in I_i has executed in the profile. This value is quantified based on the expectation collected from the program profile. Parameter $\delta(\cdot)$ quantifies the timing in clock cycles before the first critical instruction in p_j^k is reached. Finally, $H(I_i)$ is a heuristic quantifier that models the likelihood of a cache miss. For a given I-block I_i, we identify a subset of I-blocks I_w that map to the same I$ line. For each of these blocks I_w, we compute the sum $q(I_w)$ of $\alpha(b_v)$ parameters over all basic blocks b_v in I_w. Thus, we compute $H(I_i)$ as the entropy of these sums. The rationale behind this simple predictor is that cache lines that have large number of blocks that are equiprobably executed yield higher entropy and thus, higher likelihood of a cache miss. The optimization objective quantified using Eqn. 5, aims at modeling the delay due to run-time MAC verification as well as system's L1 I$ misses. It can be shown that the above problem is NP-complete because instruction (or list) scheduling, which is already NP-complete, can be reduced to it.

3.3. Optimization Algorithm

In order to address the difficult problem posed in the previous subsection, we have created an algorithm that relies on a fast most-constrained least-constraining heuristic. The prime objective behind the optimization tool is to find a basic block layout that minimizes the metric from Eqn. 5. The algorithm is described using the pseudo-code in Table 3.

```
create M empty I-blocks I_1, ..., I_M
denote I = ∪_{i=1}^{M} I_i
sort B in decreasing order of ρ
α_0 = Σ_{b∈B} α(b)
while Σ_{b∈B} α(b) < 0.95α_0
  b is basic block in B with highest ρ(b)
  while limit(t_d − δ(b)) > 0
    find block v from fan-in with minimal ρ(v)
    merge b = v||b
  endwhile
  while b is smaller than maximum I-block size
    find block v from fan-out with maximal ρ(v)
    and which is not an entry point when merged with b
    merge b = b||v
  endwhile
  find I_i that best fits b (★)
  I ← b
  remove b from B
endwhile
greedy positioning of blocks in B for best fit in I
```

Table 3. Basic block relocation.

The algorithm initially reserves memory space for the resulting instrumented program as a multiple of the size of an L1 I$ line. Next, it sorts all basic blocks in B according to the heuristic constraint model quantified in Eqn. 4. After the algorithm sets $\alpha_0 = \sum_{b \in B} \alpha(b)$, it enters a loop which sequentially processes those blocks in B which account for large percentage of α_0 (typically 90-95%). Assuming that large number of basic blocks are commonly not executed in a binary, the number of blocks that remain at this point is significantly larger than the blocks already processed.

Within the loop, the algorithm selects the most constrained basic block b according to the ρ metric. Since this block is constrained, it is likely to be of type TB or CSLB (see Subsection 3.1). In both cases, a solid heuristic is to review the fan-in of b and merge all blocks in the fan-in with b into one I-block such that paths starting from all entry points in this block have sufficient depth before a critical instruction of type (ii-iii) is executed. If this cannot be achieved for some path, then we concatenate blocks from this path until the current I-block fills. This step aims at satisfying the optimization constraint for the most constrained basic block.

If the current I-block is still not full, we can additionally concatenate blocks from the fan-out of b. Within the fan-out, we select a basic block v with the highest value of ρ and which is not an entry point itself when added to the collection of blocks already merged with b. Since all paths that lead to v are longer than t_D, then in this step we want to merge blocks of the highest available constraint from the fan-out. The intuition behind this heuristic is to resolve the most constrained components of our problem space using the least-constraining location in an I-block.

We include the hot subpaths heuristic [17] (or CAC) in this step by concatenating basic blocks until we encounter a block which contains a critical instruction that is executed for the containing I-block before the projected verification delay t_D. In that case, we undo the schedule for that I-block.

In order to optimize the relocation of basic blocks, we place the resulting collection of basic blocks b into one or more I-blocks according to the following heuristic (this step is marked in Table 3 as ★). If the average *bsed* of basic blocks in b is small, we conclude that these blocks are executed frequently in bursts. In order to avoid potential cache conflicts, blocks from b should be placed in a memory location that does not map to the same cache line as their immediate neighbors. Neighborhood is quantified with respect to the CFG, not the memory map of the original binary. In order to find potential suspect I-blocks, we identify the nodes in the CFG that are within the ε-neighborhood of the blocks in b. An ε-neighborhood is defined as a subtree of the CFG which contains nodes with a minimal shortest path to any block in b equal to ε hops. Usually ε is set within 5 and 10 hops. Conflict I-blocks are identified as the ones that contain at least one block from the ε-neighborhood of blocks in b. Blocks b are relocated to the first I-block that does not

IEEE COMPUTER SOCIETY

map to a memory location that maps to the same cache line as one of the conflict I-blocks. If conflict I-blocks cover all I$ lines, we select the first I-block that maps to a line in the I$ that has the fewest other conflict I-blocks map to it.

Once the blocks from B that account for 95% of all basic block executions are placed in their I-blocks, we apply an iterative improvement algorithm which relocates individual I-blocks such that the overall optimization goal from Eqn. 5 is improved. I-blocks are processed in decreasing order of their cost from Eqn. 5. For each I-block, the algorithm finds the most likely preceding basic block from the fan-in of I-block's entry points and the α profile. If this block can be transferred to another memory location which maps to a non-conflicting L1 I$ line such that the metric from Eqn. 5 is reduced, the algorithm indeed relocates the I-block to this position. Otherwise, it continues evaluating the next I-block. If an I-block is relocated, costs are recomputed for affected I-blocks and the algorithm backtracks in the sorted list of I-blocks to the one with the highest cost which was affected in the last step. By following this routine, the iterative improvement algorithm quickly converges to a local minimum. In the last step of the algorithm, we use the remaining blocks in B to fill the gaps between I-blocks which are only partially filled.

While the presented optimization technique is problem specific, it is flexible such that it can be adjusted for different optimization goals. In cases when an architecture facilitates "Critical Word First" cache line fetch, our platform incurs additional stall cycles. However, additional optimization constraints can be applied such that critical words appear at the beginning of cache lines. Alternatively, commutative MACs (where the order of the inputs does not affect the final result) can be applied.

4. Experiments

In this work, we have collected program profiles in-line, while executing binaries. We used two types of data collection: execution of instrumented binaries and process sandboxing. Instrumentation of binaries entails changing the binary layout with additional instructions. The role of the injected code is to divert the execution to the code that updates counters of registered events. The execution is diverted toward an external process with parameters that describe the observed event. This type of data collection is suitable for parameters that do not depend on the memory footprint, such as α, β, and γ. Instrumentation of binaries cannot be used for extracting execution parameters that are dependent upon cache behavior. These parameters include δ and χ, which have to be extracted without changing the binary layout. For such purpose, we have executed binaries in a sand-boxing environment where instructions were interpreted.

We evaluate the experimental results obtained for an implementation of our platform. The tests were run on an Intel Xeon processor at 2.8GHz. We have compiled and optimized all but five of the SPEC2000 INT benchmarks due to our inability to instrument or sand-box these programs. We also present the results for Microsoft Visual FoxPro. Static and some parts of the dynamic analysis of the main program profile parameters $\{\alpha, \ldots, \lambda\}$ for the benchmarks are presented in Table 2.

Our preliminary results presented in Table 4 quantify the performance of a system enhanced with our platform. We have run three system configurations: a traditional platform (I) optimized for L1 I$ misses using the algorithm from Subsection 3.3 denoted with ★, without optimizations (II), and with optimizations (III). We considered four L1 I$ configurations A - 16KB/128B; B - 32KB/128B; C - 16KB/256B; D - 32KB/256B, where the ratios denote L1 I$ size and its line length. Benchmarks with small L1 I$ footprints (gzip, vpr, mcf, and bzip2) resulted in negligible execution overhead due to run-time MAC verification. In order to evaluate the effect of our system on more constrained platforms, for these benchmarks we used two smaller L1 I$ configurations E - 4KB/128B; and F - 2KB/128B.

Column 2 in Table 4 shows the total number of executed instructions in billions. Columns 4, 5, 6, and 7 identify the number of L1 I$ misses, the penalty due to processor stalls in clock cycles, the ratio of penalties due to processor stalls versus L1 I$ misses, and the percentage overhead that system II incurs over system I due to processors stalls for MAC verification respectively. The next two columns labeled "First Write" and "First Exit" contain the average number of instructions executed until a critical instruction of type (*i*) for "First Write" and type (*ii*-*iii*) for "First Exit," is executed in system II. The "IBS" column displays the number of times execution path switched between different I-blocks in millions. The next two columns represent L1 D$ misses and miss percentage with a single L1 D$ configuration: 8-way associative 32KB with 128B cache lines. The last two columns represent the overhead achieved by system III with respect to system I and thus, the improvement of our code optimization techniques versus the overhead of the non-optimized system II respectively. One can observe that the normalized average overhead reduction totalled 60.4% resulting in maximal overall overhead of 11.25% in only one case. In the majority of other cases, the overhead was negligible. The prototype optimization procedure presented in Section 3 executed in less than 10 seconds for each program on an Intel Xeon processor at 2.8GHz.

5. System Security

It is important to stress that our platform does not prevent nor detect buffer overruns. It aims at preventing the adversary from running a single line of her own code on a protected machine. Our platform forces the adversary to use a buffer overrun exploit to jump into binaries that are already running in protected mode and feed them with de-

IEEE COMPUTER SOCIETY

SPEC	Exec. instr. $\times 10^9$	Cfg.	I$ misses	Penalty	Pen. /mis.	Ovhd. [%]	First Write	First Exit	IBS x10^9	D$ misses x10^6	D$ miss rate [%]	I.O. [%]	Imp. [%]
gzip	515	A	403 x10^3	5167 x10^3	12.8	0.01	12.3	6.5	47.0	6885	1.34	0.00	11.3
		B	130 x10^3	1582 x10^3	12.1	0.00						0.00	9.2
		C	347 x10^3	6525 x10^3	18.8	0.00	14.9	9.0	36.1			0.00	11.7
		D	4 x10^3	66 x10^3	17.9	0.00						0.00	10.0
		F	3170 x10^6	58 x10^9	18.2	11.2	15.1	10.1	47.0			6.87	38.7
vpr	138	A	54 x10^3	702 x10^3	13.1	0.01	8.1	6.6	10.8	1665	1.20	0.00	93.7
		B	5 x10^3	66 x10^3	13.1	0.00						0.00	18.8
		C	18 x10^3	431 x10^3	23.7	0.00	11.8	6.1	7.5			0.00	88.4
		D	4 x10^3	66 x10^3	16.9	0.00						0.00	29.5
		E	721 x10^6	15 x10^9	20.8	13.23	6.9	6.9	10.8			5.99	54.7
mcf	112	A	1498	19025	12.7	0.00	12.8	6.4	12.5	4612	4.12	0.00	11.9
		B	896	12275	13.7	0.00						0.00	11.2
		C	1293	24696	19.1	0.00	15.7	6.9	8.2			0.00	70.2
		D	491	9722	19.8	0.00						0.00	9.5
		F	2 x10^9	36 x10^6	17.2	32.34	8.9	5.8	12.5			11.25	68.0
perlbmk	523	A	1073 x10^6	15 x10^9	14.2	2.91	5.7	9.8	38.8	640	0.12	1.54	47.1
		B	620 x10^6	9 x10^9	14.6	1.73						0.82	52.6
		C	1071 x10^6	215 x10^9	19.2	10.74	6.7	12.8	26.7			4.61	57.1
		D	554 x10^6	115 x10^9	19.9	7.19						2.03	71.8
vortex	439	A	3153 x10^6	35 x10^9	11.0	7.90	6.3	7.1	42.9	1198	0.27	3.39	57.1
		B	875 x10^6	10 x10^9	10.9	2.17						1.01	53.5
		C	2075 x10^6	34 x10^9	16.4	7.75	8.5	11.2	20.7			3.04	60.8
		D	828 x10^6	13 x10^9	16.1	3.03						0.99	67.3
bzip2	526	A	32298	413414	12.8	0.00	13.0	6.6	34.4	2457	0.47	0.00	13.2
		B	2791	35725	12.8	0.00						0.00	14.1
		C	16281	333761	20.5	0.00	15.1	8.0	22.2			0.00	14.5
		D	1668	35361	21.2	0.00						0.00	17.0
		E	1323 x10^6	23 x10^9	18.9	4.43	11.2	5.2	34.4			2.13	43.2
twolf	476	A	671 x10^6	9 x10^9	13.4	1.93	5.1	6.9	24.1	5957	1.25	0.36	81.3
		B	190 x10^6	3 x10^9	13.9	0.55						0.23	58.2
		C	484 x10^6	11 x10^9	22.7	2.31	9.9	8.1	20.0			0.19	91.7
		D	148 x10^6	4 x10^6	23.8	0.74						0.12	83.7
Visual FoxPro	8	A	114 x10^6	854 x10^6	7.5	10.66	13.1	13.8	0.77	16	0.20	5.06	52.5
		B	39 x10^6	203 x10^6	5.2	2.53						1.48	41.5
		C	115 x10^6	776 x10^6	6.8	9.68	14.9	11.0	0.58			5.23	46.0
		D	44 x10^6	278 x10^6	6.3	3.47						2.81	19.0

Table 4. Cache statistics for SPEC2000 benchmarks with four I$ size / I$ line length configurations: A - 16KB/128B; B - 32KB/128B; C - 16KB/256B; D - 32KB/256B. For benchmarks with small L1 I$ footprints (gzip, vpr, mcf, and bzip2) we include two more configurations: E (4KB/128B) and F (2KB/128B). Columns 2 contains the total number of executed instructions. Columns 4, 5, 6, and 7 contain number of L1 I$ misses, secure execution penalty, ratio penalties over L1 I$ misses, and total execution overhead incurred by the penalty. The next two columns labeled "First Write" and "First Exit" contain average indexes of the first encountered write instruction in the L1 I$ line and average index of the first instruction that causes switch of L1 I$ lines. The "IBS" column displays the number of times execution path switched between IBs. Fourth to last and third to last columns represent L1 D$ misses and miss rates with a 128B CL, 8-way associative 32KB D$. The last two columns represent the performance overhead and improvement yielded by our optimization algorithm.

sired data to perform malicious actions. By (re)loading programs at random locations in operating memory, this task can be made difficult. Thus, our platform significantly reduces the likelihood and functionality of attacks an adversary can launch against a protected system. Techniques that prevent or detect buffer overruns (e.g., StackGuard [5]) can be used in conjunction with our platform.

Our platform provides robust system security, only if certain conditions in the OS kernel mode are met. Namely, system penetration can occur if the OS kernel mode is interpreting instructions or scripts. Scripts or virtual machine binaries are treated as data, not code; hence, they cannot be authenticated using the mechanisms presented in this paper. In order to enable the kernel to run scripts, the OS can provide an associated script mode with limited access to system resources and in particular, prohibitive policy for calling the software installer from this mode. The OS in this mode can perform script authentication before execution either via public-key signatures or proof-carrying code [18]. However, there are no guarantees that the script is not vulnerable to a buffer overrun attack.

Our platform has physical limits in protecting computing systems. It cannot survive physical breaches into the computing system. For example, a detached hard-drive exposes installed binaries which can be used to create small malicious programs by patching signed I-blocks from existing working-copies. The resulting programs can be made to infiltrate both the OS and other users' accounts. Note that the patching attack can be made difficult by encrypting certain key storage containers of the OS. A sophisticated adversary should be able to expose all secrets in the system by thoroughly reverse engineering both hardware and software. For an adversary who is operating remotely, the patching attack is impossible according to the definition of intrusion prevention systems.

6. Related Work

Four major approaches for code security have emerged: code signing, sandboxes, firewalling, and proof-carrying

IEEE COMPUTER SOCIETY

code. Signing a program binary for authentication purposes is conceptually the simplest code security technique. In this case, authentication is done according to standardized authentication protocols [1]. 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 [21]. The main idea behind the firewalling technique for code security is to conduct comprehensive examination of the provided program at the very point where it enters the consumer domain. Necula 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 distrusted source [18]. This is accomplished by requesting that the source provides a security proof that attests to the code's adherence to a host defined security policy.

The developed attempts to detect or prevent 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 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 [6]. XOM, a platform for tamper- and copy-resistant software has been developed [9] to create the kind of memory that is exclusively dedicated to program binaries, that could not be read or modified, but only executed. XOM decrypts program binaries at run-time and utilizes hardware-supported memory management to achieve tamper-resistance. In a sense, XOM is the closest relative to both SPEF (described in Subsection 1.1) [8] and our platform.

Static source code analysis techniques range from fast and simple error detection, such as type errors and uninitialized variables [15, 22] to complex and relatively slow formal verification-based tools that detect variety of bugs, including null pointers and errors in definitions, allocation and aliasing [7, 12, 11]. Wagner et al. used static analysis techniques as the first step toward automated buffer overflow detection – their static analyzer commonly generates false alarms that must be verified manually [10].

7. Conclusion

We have introduced a novel, simplified, hardware-assisted intrusion prevention platform with a focus on creating techniques that significantly reduce the performance overhead incurred due to run-time MAC verification. We have proposed a software optimization technique that initially identifies critical instructions, ones that are likely to fill the buffers dedicated to support speculative execution, and then reorders basic blocks within a given block so that critical instructions are executed as late as possible within an instruction block in common cases. We have conducted experiments by executing the SPEC2000 benchmark on a traditional and x86 platform enhanced with our system. Preliminary results show that our code optimization techniques produced an overhead reduction of up to 90%.

References

[1] A. Menezes et al. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, FL, 1996.

[2] ASICS.ws. Enhanced AES (rijndael) IP core. *Available online at http://www.asics.ws*.

[3] J. Bradford and R. Quong. An empirical study on how program layout affects cache miss rates. *ACM SIGMETRICS Performance Evaluation Review*, 27(3):28–42, 1999.

[4] T. Burd. CPU info center. *Available on-line at http://bwrc.eecs.berkeley.edu/CIC*.

[5] C. Cowan et al. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. *USENIX Security Symposium*, pages 63–77, 1998.

[6] S. Chari and P.-C. Cheng. Bluebox: A policy driven, host-based intrusion detection system. *NDSS*, 2002.

[7] D. Evans et al. LCLint: A tool for using specifications to check code. *ACM SIGSOFT*, pages 87–96, 1994.

[8] D. Kirovski et al. Enabling trusted software integrity. *ASPLOS*, pages 108–120, 2002.

[9] D. Lie et al. Architectural support for copy and tamper resistant software. *ASPLOS*, pages 169–177, 2000.

[10] D. Wagner et al. A first step towards automated detection of buffer overrun vulnerabilities. *NDSS*, pages 3–17, 2000.

[11] U. S. et al. Detecting format string vulnerabilities with type qualifiers. *USENIX Security Symposium*, pages 201–220, 2001.

[12] D. Evans. Static detection of dynamic memory errors. *ACM PLDI*, pages 44–53, 1996.

[13] G. Memik et al. Just say no: Benefits of early cache miss determination. *HPCA*, 2003.

[14] J. Torrellas et al. Optimizing instruction cache performance for operating system intensive workloads. *IEEE HPCA*, pages 360–369, 1995.

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

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

[17] J. Larus. Whole program paths. *ACM PLDI*, pages 259–269, 1999.

[18] G. Necula. Proof-carrying code. *ACM POPL*, pages 106–119, 1997.

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

[20] D. Seeley. The internet worm, password cracking: a game of wits. *Communications of the ACM*, 32(6):700–703, 1989.

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

[22] C. Wilson and L. Osterweil. Omega - a data flow analysis tool for the C programming language. *IEEE Transactions on Software Engineering*, 11(9):832–838, 1985.

1072-4451/04 $20.00 © 2004 **IEEE**

IEEE COMPUTER SOCIETY