Proceedings of the 2005 Winter Simulation Conference
M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, eds.

PERFORMANCE ANALYSIS OF BINARY CODE PROTECTION

David M. Nicol
Hamed Okhravi

Dept. of Electrical and Computer Engineering, and
Coordinated Science Laboratory
University of Illinois, Urbana-Champaign
Urbana, IL 61801, U.S.A.

ABSTRACT

Software protection technology seeks to prevent unauthorized observation or use of applications. Cryptography can be used to provide such protection, but imposes a potentially significant additional computation load. This paper examines the performance impact of two software protection techniques. We develop an analytic model and validate it using a detailed discrete-event simulator applied to memory reference traces of well-known benchmark programs. We find that even though the added workload may be large, that impact is often dominated by inherent costs of disk activity.

1 INTRODUCTION

The field of computer security has largely focused on networks, and controlling access to systems. There is burgeoning interest through in securing *applications*. The concern is that application executables reflect a significant amount of intellectual content (e.g. the algorithms they execute) that may be sensitive. Consequently, in addition to protecting access we would like to protect the application itself against unauthorized observation, analysis, or execution.

There is an ever-present tension between degree of protection and ease of use and implementation. Many commercial programs require a serial code in order to install the program, yet make no attempt to check that the user does not install the program on more machines than the license permits, or prevent a knowledgeable user from creating an unauthorized copy elsewhere by replicating the right set of files. These programs are easy to use, but have easy to defeat defenses. A stronger degree of protection is offered by combining some machine specific information with a key, in a way that is protected from an adversary. Certainly at installation time (and with more work, at run-time) the system may enforce authorization constraints. However, this degree of protection requires support mechanisms to protect the acquisition of machine ID and combination of that ID with key information.

A variety of techniques have been proposed and are being investigated under the auspices of the *Software Protection Initiative* (Clark 2003). Broadly speaking, SPI aims to develop technologies to combat software piracy, to insure that code has not been altered in an unauthorized way (e.g. to insert monitors or Trojan horses), and to ensure that when an application executes it does so only by an authorized user. Some of the goals are short-term, aimed at protecting existing applications without modification. Longer term goals allow protection technologies to become part of the software development process.

This paper asks what the impact on performance might be by adopting certain approaches that support SPI objectives. One protection approach hardens the virtual memory system, ensuring that no part of the program is ever exposed in plaintext on disk, even while executing. A more severe protection approach hardens the physical memory system, ensuring that no part of the program is ever exposed in plaintext in main memory. As one would expect, there is a protection/performance tradeoff to be assessed. Our approach is to develop a model focused on execution time. We validate the model using simulation. The simulator takes as input traces of references to memory observed in SPEC 2000 (SPEC 2000) benchmark programs. The simulator introduces delays as a function of hits and misses in the cache, hits and misses in the virtual memory system, and disk delays. The model with protection introduces the additional delays. We examine the impact on the execution time of a single program, and the increase in native computational effort. We find that a program running under the hardened virtual memory system suffers low relative impact on execution time almost regardless of its locality of reference characteristics, simply because the delay due to paging is high relative to execution costs. However, the relative amount of extra work done to provide the protection *is* sensitive to locality of reference. Turning to the techniques

for hardening physical memory we see a much increased impact on execution time and computational work devoted to protection. However, we observe that these impacts are ameliorated, in part, when multiple such protected programs are executed concurrently in a multi-tasking environment.

2 PROTECTION OF BINARIES

The area of SPI concern that is of interest to us is the protection of software binaries. Specific objectives include detecting unauthorized modifications, detecting unauthorized reverse engineering (e.g. through unauthorized use of tools that analyze binaries, such as disassemblers), detecting when the operating environment for the application is not what is expected, and protecting memory and the file systems. Some of these objectives are approached by obfuscating binary executables in such a way that even if the binary is obtained by a hostile agent, analysis of that binary will not reveal sensitive application information, at least not without an enormous amount of effort.

The literature on obfuscation is large, some recent examples include (Naumovich and Memon 2003), (Collberg and Thomborson 2002),
(Stytz and Whittaker 2003) and their references. The basic idea to make changes in a textual form (e.g. source or assembly) of a program that hide the author's intent, but lead to a functionally equivalent program. Several different aspects of a program might be so obscured. The object code generated by a compiler contains numerous strings used to label and name locations and variables. One means of obscuring meaning is to substitute nonsense strings for meaningful ones, such as is done in the Shroud system for C programs (Jaeschke 1990). One can also rearrange the location of data, and the code used to access it. For example one could transform the sequential scan of an array into a permuted access by reordering the way the array is laid out, and inserting code that generates the correct sequence of array indices in place of the sequential scan. The control logic of a program can be changed, e.g., by identifying statements that can be executed out of the expressed order, and changing the order. One can also obfuscate code by removing portions of it altogether, to be replaced by remote procedure calls to a server that provides the necessary functionality.

One of the attractive features of obfuscation is that it is self-contained: the operating system does not need to be modified, there is no requirement for cryptographic key management. From an information theoretic point-of-view obfuscation is a weaker form of protection than strong encryption. It also requires obfuscation tools that are specific to the language (e.g. C++ or Java). Knowledge of the techniques used to obfuscate a program can be used to try and de-obfuscate it.

Key management *is* an issue when binaries are encrypted. Different contexts and security requirements call for different approaches. One context requires that the host machine be oblivious to the fact that the binary is encrypted, and is being decrypted. This means that the binary must contain a plaintext decryption engine that controls decryption and execution of the coded binary.

Interestingly, some of the work in binary encryption comes from the hacker community. One of the motivations is to find ways of encrypting code, decrypt it as it executes on a hacked machine, and leave behind as little trace as possible. Techniques such as that described in (grugq and scut 2001) and (Mehta and Clowes 2003) embed a plaintext decryptor in the program (but use techniques of code obfuscation to mask its operation), modify the program entry point to be this decryptor, and then encrypt portions of the executable. Once inserted on a machine and run the decryptor executes once to decrypt the entire program *in memory*, and runs it. This leaves the encrypted version on disk, and the decrypted version in main memory (to be released when the program terminates). Whatever key is needed to decrypt has to be hidden, accessed, or reconstructed by the decryptor.

Other sophisticated techniques for binary protection are described in (Griffiths 2005). One can minimize the exposure of decrypted executables by marking the permissions of each page in such a way as to cause an exception to be thrown whenever a first access to the page is made. At startup the program registers a signal handler with the operating system to be executed when such an exception occurs. The signal handler can decrypt the page just touched, and re-encrypt any pages it likes that are in the memory in plaintext form. Such a technique can reduce the number of plaintext pages in memory to just 1 (albeit with a potentially high overhead). The paper goes on to describe a technique called "running line" that decodes a program one instruction at a time. The intent of such techniques is to defeat analysis by debuggers.

The context is a little different when the protected program is intentionally run on the host. Indeed, the program may be written to run only on that host, or only on hosts that provide it with some functionality not evident in the binary. For example, hardware support for security is appearing, and it will soon be commonplace for CPUs to have protected areas where secret keys may be stored, and/or cryptographic operations may be performed. In this discussion we'll call that a "cryptographic function area", or CFA. In this context a binary might be encoded by a server specifically for a given machine, with a given verifiable fingerprint. For example, a public-private key pair may be associated with the hosts CFA, with the private key being in CFA memory protected from access by any software on the host. Likewise, a binary encryption server could have a public-private key pair. A host that wants a protected binary can use the server's public key to encrypt and send some fingerprint constructed from

the identity of the host. The server uses that fingerprint to generate a symmetric key for encrypting the binary, encodes that key using the CFA's advertised public key, and sends it to the host, which passes it to the CFA, where the symmetric key is decoded and stored. The encrypted binary follows. The host can ask the CFA to decrypt (and re-encrypt) blocks of the binary using the symmetric key that is hidden in the CFA.

In this latter context then correct operation of the encrypted program requires support from the host. Security is enhanced, at the price of requiring key management and specialized hardware. These are acceptable costs for SPI, *particularly* if these can be used to protect legacy codes, without changing those codes. This paper develops a performance analysis of such solutions.

3 PROTECTION BOUNDARIES

Assuming that an encoded executable can be securely delivered to a host, the issue remains of protecting it at the host. Access is one dimension of the threat. Figure 1 illustrates different access boundaries, and the protection mechanisms sufficient for each. The strongest boundary assumes that files cannot be observed once resident in the place of installation, we need only be concerned about protecting the program by encryption while being transported to the installation. A weaker boundary admits adversarial access to program files on disk, but denies access to main memory. For example, this threat model allows for an adversary to get (either by physical access or electronic penetration) a copy of an executable from the disk. We analyze a scheme that protects against this possibility by ensuring that every representation of the program and its data on disk are encoded. A weaker boundary exists if an adversary has physical access to the computer and can trace traffic between CPU and memory using probes. Protection against this threat must obscure executable program pages and data that are resident in main memory, and must harden the process that interprets these. Such measures require sophisticated hardware and/or software. For example, the executable program might be expressed in a form that is philosophically related to Java byte-code (in-so-far as it is an implementation-independent representation)—but obscured—and have the running application implement a virtual machine interpreter.

The first protection mechanism we model hits a sweet spot in the spectrum of threats and costs of protecting against them. It protects the program binary and data up to the physical memory hierarchy, does so using technology that will soon be widely available, and is applicable to legacy software. The central question we address asks what impact this protection has on performance. The delay cost is incurred only at page faults, and is largely subsumed by the cost of handling a page

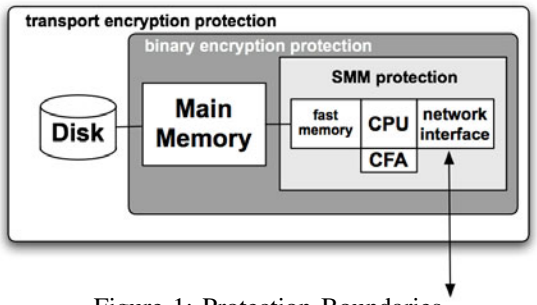

Figure 1: Protection Boundaries

fault. We expect—and our simulations confirm—that the performance impact is low. The second mechanism we model is of a "Scratchpad Memory Manager" (SMM), that decodes, caches, and executes instructions without ever making the decoded instructions or data visible to main memory in plaintext. This mechanism assumes a "scratchpad memory" (Banakar et al. 2002), (Avissar et al. 2002), (Ravindran et al. 2005) that supports software control of what enters and exits this small CPU-speed memory, and when. This type of architecture is of considerable interest to the embedded computing community, who need such control to make predictable the execution time of real-time applications.

Our principle objective is to assess the performance impact of these two protection mechanisms, in terms of program execution time. We develop a model of program execution costs, as a function of program locality and system parameters. We validate the model with a trace-driven simulator, using memory traces of SPEC benchmark programs as input. The model accounts for delays introduced by cache management, virtual memory, and encoding/decoding. We estimate the performance impact by comparing the execution time of the reference program with, and without protection, as a function of the program's locality of reference in the cache, and in the main memory. We find that when the cache locality of reference is good, the first protection mechanism performs as well as that of an unprotected program, regardless of the program's locality in main memory. We see that the second protection mechanism can have much higher processing costs, but that those costs can be dominated by significantly larger disk processing costs, or significant levels of multiprogramming. Thus we see that in ordinary contexts both mechanisms can provide protection without undue impact on performance.

4 BASIC SYSTEM MODEL

We use a system model that focuses on the memory translation process. Our model contains a cache, a main memory, and a disk-based virtual memory. The program execution is described in terms of a trace of memory references, from programs in the SPEC 2000 benchmark suite. Given a memory address, the simulator seeks it in the fast memory

adjacent to the CPU. In one set of experiments we assume that memory is organized as a 4-way set-associative cache, in another set we assume it is scratchpad memory that holds entire pages.

We assume that the effective time per instruction is 0.3 ns (excluding cache misses), and that on average one out of four instructions is a memory reference. The cost of a miss (in the cache-based simulations) includes writing a replaced line back to memory (if it is marked as dirty), and also includes the cost of reading a cache line from memory. The cost of transferring a 128 byte line between cache and memory is 63 ns, based on a bus standard of DDR266, which has a data rate of 2.1GBps.

A reference to main memory is checked to see if the referenced page is resident. If not we sample a random delay to represent the cost of going to disk. For a write of a dirty page, or a read of page that is not first preceded by the writing of a dirty page, we assume that the disk heads could be anywhere, with equal probability. We assume that the swap space is at one end of the cylinder range, and so model the number of cylinders (e.g., distance) over which the heads must pass to align to the target cylinder as a uniform integer random variable sampled in the range $[1, N_c]$, where N_c is the number of cylinders. To translate this distance into a delay we model the seek time using the model described in (Lee and Katz 1993). Specifically, we take the delay of moving the disk head $n > 0$ cylinders to be $d(n) = a\sqrt{(n-1)} + b(n-1) + c$, where, if s_{min}, s_{avg}, and s_{max} are the minimum, average, and maximum seek times (respectively), then

$$\begin{aligned} a &= \frac{-10s_{min} + 15s_{avg} - 5s_{max}}{3\sqrt{N_c}} \\ b &= \frac{7s_{min} - 15s_{avg} + 8s_{max}}{3N_c} \\ c &= s_{min}. \end{aligned}$$

Our simulations use parameter values modeled after a 80Gb Western Digital WD800BB drive : $s_{min} = 2ms, s_{avg} = 9ms$, $s_{max} = 21ms$, and $N_c = 16383$.

The second component of disk delay is due to rotation, waiting for the correct sector to align with the heads. Assuming a 7200 rpm disk, we model this delay with a uniform random variable sampled over [0,1/7200] seconds. For the case of a disk read that immediately follows the writing of a dirty page we assume the heads are “close enough", and assume only a random rotational delay. We assume the disk has a large cache buffer, and that 60% of the read requests are satisfied from this buffer. The final component of disk delay is transferring the disk block to the main memory, which we assume is done using a 100Mbs bus.

5 ENCRYPTION AND CONTROL

The first protection mechanism we consider never leaves program pages in plain-text on a disk. At its point of origin the program executable is divided into pages, each of which is individually encrypted. The encryption is 'symmetric', meaning that the same key K used to encrypt the pages is used to decrypt them. The key used can be generated in such a way that use of it is restricted to a particular machine (or sets of machines). This customarily is accomplished by having hardware support on the machine that executes the program that binds the key to some unique physical characteristic of that machine (e.g. a secure coprocessor holding private keys associated with asymmetric cryptography. The symmetric key K can be conveyed after encoding by the coprocessor's public key; only the coprocessor can obtain the symmetric key.) The encryption / decryption costs assumed by our simulations are based on the the AES algorithm (Daemen and Rijmen 2002), used with a 128 bit key. Our experiments assume an execution cost of cost of encrypting or decrypting an N-Kb page as $36N\mu$-secs; this figure comes from measurements of a publicly available optimized ANSI C version of the algorithm, scaled to execute on a 4GHz machine.

The most straightforward implementation of this mechanism involves a small modification to the host computer's operating system. The modifications needed are not difficult for the open source Linux operating system. The main idea of the approach we analyze is to place the encryption/decryption logic into the virtual memory management. Different pages may come from different sources (e.g. application, code from various system libraries, data files) and be encrypted using different keys. The data structures that manage virtual memory mappings need to be augmented to record, for each page, some index to the appropriate encryption/decryption key. (As we pointed out earlier, the key itself may be protected within an CFA; we assume here an interface that allows the operating system to request an encrypt or decrypt operation using a specific key by specifying the index.) Extra logic is needed in the virtual memory system to decrypt incoming pages before they are used, and encrypt pages that are about to be written back to swap space.

The virtual memory system works almost exactly as before. A page fault causes an interrupt, and the operating system looks for a free page frame into which it might place the needed page. If it must eject a page that has been modified, that page will be written out to the swap area. Here we require a modification that has the operating system first ask the CFA to encrypt the page, in-place, in memory, and *then* write it out to disk. The augmented page table will specify the index of the key needed to perform the encryption. Now when the disk has completed the fetch of a requested page, it raises an interrupt. Taking the interrupt,

Table 1: Variables Used in Analytic Model

p_m	Pr{main memory hit}
p_{dm}	Pr{reference evicts dirty page}
p_c	Pr{cache hit}
p_{dc}	Pr{reference evicts dirty cache line }
p_s	Pr{scratchpad hit}
p_{ds}	Pr{reference evicts dirty scratchpad page}
t_{inst}	time to execute instruction
t_{line}	cache transfer time, 128 bytes
t_{aes}	AES encrypt/decrypt time, 1K bytes
n_{page}	number of 1Kb per page
t_{sp}	transfer time, 1Kb, scratchpad
t_{seek}	average disk seek time
t_{rot}	average disk rotational delay
t_{trns}	disk transfer time per 1Kb block
n_m	number of page misses in trace
n_{dm}	number of dirty memory pages evicted
n_c	number of cache misses in trace
n_{dc}	number of dirty cache lines evicted
n_s	number of scratchpad misses in trace
n_{ds}	number of dirty scratchpad pages evicted
n_{ir}	number of instructions per memory reference
n_{ref}	number of references in the trace

the operating system calls upon the CFA to decrypt the page, after which the virtual memory logic is exactly as before. A principle benefit of such an approach is that legacy programs and libraries can be protected without any modification. The principle disadvantage is that it requires modification to the operating system, albeit a small modifications, which may be problematic when the operating system source code is not available.

The second protection mechanism we consider goes through similar steps. We assume that the scratchpad memory is used to hold entire pages, that any page copied from main memory to the scratchpad is in an encrypted state, and that such a page is decrypted and held in the scratchpad. On a scratchpad fault, if a page must be evicted (to main memory) it must first be encrypted. Main memory now serves primarily as a cache buffer between scratchpad and disk memories; of course, the usual virtual memory actions (and costs) still apply. We have just moved the encryption / decryption costs to the fast memory and made different assumptions about how memory references are interpreted in that fast memory.

6 ANALYSIS

We have developed models for predicting the average execution time per memory reference of protected binaries. Parameters of these models are explained in Table 1. All of these probabilities are unconditional, each represents the probability that a memory reference causes the supposed action.

First consider the baseline case, where no protection is used. The average CPU time per memory reference is

$$\mu_{cpu} \quad = \quad n_{ir} * t_{inst} + (\bar{p}_c + p_{dc}) * t_{line}.$$

This expression accounts for processing instructions, transferring a cache line on every miss, and transferring a cache line when an evicted line is dirty. The average additional delay due to disk operations is

$$\mu_{disk} = \bar{p}_m * (t_{seek} + t_{rot}) + p_{dm} t_{rot}.$$

This expression encodes the assumption that no head movement is needed between writing an evicted dirty page to disk and reading the page that evicted it. Now the cost of protecting virtual memory is the cost of encrypting or decrypting pages moving between memory and disk. This additional cost is thus

$$\mu_e = (\bar{p}_m + p_{dm}) * t_{aes} * n_{page}.$$

The execution slowdown due to protection is the ratio

$$(\mu_{cpu}+\mu_{disk}+\mu_e)/(\mu_{cpu}+\mu_{disk}) = 1+\mu_e/(\mu_{cpu}+\mu_{disk}).$$

It is not difficult to see that the slowdown will be small whenever $\mu_e \ll \mu_{disk}$, regardless of the cache and main memory hit rates. We have measured the former cost on the order of 100 μ-secs, specifications for modern disks estimate the latter cost on the order of 10 $msec$, a difference of two orders of magnitude. By contrast, the factor by which CPU workload increases using protection is the ratio

$$(\mu_{cpu} + \mu_e)/\mu_{cpu} = 1 + \mu_e/\mu_{cpu}.$$

Typically the timings that quantify μ_{cpu} are measured in nano-seconds, whereas t_{aes} is measured in micro-seconds. The value of this ratio depends significantly on cache and memory hit probabilities, and can be large when locality of reference is not good.

A model for the average execution time when protecting physical memory is similar. CPU cost associated with this protection scheme is

$$\gamma_{cpu} \quad = \quad n_{ir} * t_{inst} + (\bar{p}_s + p_{ds}) * n_{page} * t_{sp}.$$

The average overhead associated with cryptography is

$$\gamma_e = (\bar{p}_s + p_{ds}) * n_{page} * t_{aes}.$$

The slowdown due to protection mechanisms in this case is $1 + \gamma_e/(\mu_{cpu} + \mu_{disk})$. This overhead is sensitive to cache

locality, for the cost of a miss is considerably higher than in a normal system. The factor by which the CPU load increases is $(\gamma_{cpu}+\gamma_e)/\mu_{cpu}$. This is orders of magnitude larger than the corresponding factor under the other protection scheme.

We validated these formulations by writing a functional discrete-event simulation of the model described in Section §4. The simulation takes memory reference traces as input. It explicitly simulates the identity of values in the cache (and in the scratchpad), it manages explicitly the identity of pages in the virtual memory. In order to provide a realistic context of background memory demand, we measured memory usage over extended periods of time in computer running Linux, and varying the amount of physical memory available to our simulated program accordingly, as it competes for resources represented by the background memory consumption. Thus this simulator captures with some detail the impacts of architecture and operating context on the behavior of the program. The output of this simulator is an enumeration of memory activities—the number of cache hits/misses, the number of dirty cache line evictions, the number main memory hits/misses, the number of dirty memory page evictions, and so on. These counts completely determine the costs of transferring data between memory hierarchies; since the protection mechanisms are all tied to data movement within the memory hierarchy, the summary serves to determine those costs as well. The action summary is read by a separate timing simulator. Many of the time delays are deterministic (at least given the level of detail in this model). The component of execution time related to non-disk activities in the caching system is

$$T_c = n_{ir} * n_{ref} * t_{inst} + (n_c + n_{dc}) * t_{line}.$$

The corresponding component for the scratchpad system is

$$T_s = n_{ir} * n_{ref} * t_{inst} + (n_s + n_{ds}) * n_{page} * t_{sp}.$$

The work associated with cryptography in the caching system is

$$T_{cc} = (n_m + n_{dm}) * t_{aes} * n_{page},$$

while the work associated with cryptography in the scratchpad system is

$$T_{sc} = (n_s + n_{ds}) * t_{aes} * n_{page}.$$

However, our model does assume random costs associated with the disk, due to uncertainty in the distance that the heads must be moved, and the rotational latency. A distinct advantage of separating the functional memory simulator from the timing simulator is that one study the impact of different timing costs (e.g. bus speeds, disk technology). One can sample random disk delays multiple times, enabling us to study the sensitivity of execution time to this stochastic component. The timing simulator therefore estimates the mean disk delay (according to the disk model already specified), and reports the standard deviation for a given number of samples.

The data reported by the functional simulator for a given trace can be used to construct parameters for the analytic model. Specifically

$$\begin{aligned} p_m &= 1 - n_m/n_{ref} & p_{dm} &= n_{dm}/n_{ref} \\ p_c &= 1 - n_c/n_{ref} & p_s &= 1 - n_s/n_{ref} \\ p_{dc} &= n_{dc}/n_{ref} & p_{ds} &= n_{ds}/n_{ref}. \end{aligned}$$

By construction, use of these estimators in the analytic model will cause the deterministic components to fit the measured data exactly. The behavior of the disk component is more interesting. The total delay attributed to the disk is the sum of a number of independent identically distributed random numbers. The precise number is driven by the program's locality behavior; in our evaluations of traces with 10^9 memory references, a memory miss rate of, say, 10^{-6}, yields on the order of a thousand disk accesses. The central limit theorem obviously applies then, so that given D disk accesses, the total disk cost is approximately normal $N(D\mu_d, \sqrt{D}\sigma_d)$, where μ_d and σ_d are the mean and standard deviation of the random disk delay. Computing the details of μ_d and σ_d is more technical than it is interesting, but there is a very important point to be made. σ_d/μ_d cannot be large, because the disk seek time distribution does not have a large range relative to its mean, the rotational latency is essentially just a uniform random variable, and the probabilistic choice based on whether the page is in cache or not is fairly balanced. This all implies that the coefficient of variation $\sqrt{D}\sigma_D/(D\mu_D) = (\sigma_D/\mu_D)/\sqrt{D}$ will be small in our application so that the analytic model (which uses μ_d) is bound to predict results close to those observed in any long-lived experiment.

We evaluated two different traces taken from executions of two particular SPEC 2000 benchmark suite. Both are representative of the the kinds of codes that might merit protection. *lucas* performs a test (the Lucas-Lehmer test) to check the primality of Mersenne numbers $2^p - 1$, using arbitrary-precision arithmetic. Computation of large squares is based on an FFT designed to be cache friendly. *sixtrack* is a high-energy physics code that tracks particles around a model of particle accelerator, in order to check the long term stability of the particle beam.

We simulated the execution of these codes assuming a host computer with 0.5Gb of memory, a 4GHz CPU, 1Mb of fast CPU speed memory (alternatively treated as a 4-way set associative cache with 128 byte lines, and a scratchpad memory with page-size units of transfer between it and main memory. We assume a 2.1GBps bus between fast memory and main memory, and a 100Mbs bus between main mem-

ory and the hard drive cache buffer. We assume that 60% of disk accesses are resolved in the hard drive cache. For each experiment we simulated one billion memory references. As we expect, comparison of all measured costs and predicted costs (base execution, encryption, disk) costs were extremely close; the relative error between predicted and measured costs were in tiny fractions of a percent. Table 2 presents the locality behavior of these traces on the simulated architecture, and tabulates the various average costs per memory reference, assuming both 2K and 4K pages. Prominent features of this data include that the programs enjoy good locality of reference, that disk costs essentially halve going from 2k to 4k pages, and that encryption costs increase by 50% going from 2k to 4k pages. It is also very noteworthy that the overall execution time is dominated by the disk delay.

We have shown why the analytic model is an excellent predictor of long-term behavior of any program. We now use that model to explore regions of behavior not exercised by the benchmark traces. We compute two figures of merit. One is based on estimated execution time, without any cryptographic protection, and with protection. Under the assumption that the system is essentially dedicated to the program (so that the program never waits for CPU service when it is able to execute after processing of a page fault, and that it never waits for disk service upon processing a page fault) we can compute the execution slowdown—the factor by which the program completes more slowly using protection than not. As seen in Table 2, under all circumstances tested, both protection techniques yield a very small slow-down. The reason is that the execution time is far and away dominated by the common cost of disk access. To illustrate that point, we compute also the ratio of the execution time component *except* for the disk. This ratio gives the factor by which protection mechanisms require more CPU and fast memory resources than not.

These experiments show that under some circumstances one can use cryptographic protection without suffering undo performance penalties. The space of possibilities probed by these traces is small. But with the confidence we've gained in our analytic model, we can explore a much wider space of possibilities. Some of the results of that exploration are illustrated in Figure 2. Here we consider the main memory miss rate to of the same order as those observed in the SPEC benchmarks), and evaluate performance as a function of the fast memory miss rate. Figure 2 shows the remarkable tendency of the execution time ratio associated with the virtual memory protection scheme to be small, independent of locality of reference. The explanation is just that cryptographic costs occur precisely when disk operations supporting virtual memory occur, and those cryptographic costs are overwhelmed by the inherent delays in accessing disk. Changing the locality of reference characteristics changes the frequency of disk accesses, but the frequency of cryptographic costs changes in lock-step. The impact on the mechanism for protecting physical memory is more marked. Here cryptographic costs occur as a function of locality of reference in the scratchpad memory, so the frequency of incurring those costs increases as the scratchpad miss rate increases. We also computed these same merits for a family of applications whose underlying memory miss rate is an order of magnitude larger. The comparison shows that as scratchpad locality degrades, the application with an order of magnitude smaller main memory miss rate incurs an order of magnitude larger slowdown. This is understood by observing that when scratchpad locality is bad the execution time is dominated by the cryptographic costs of moving things into and out of the scratchpad memory. The denominator of the execution time ratio is an order of magnitude smaller when the volume of disk activity is an order of magnitude smaller, which occurs when the main memory miss rate is an order of magnitude smaller. These graphs clearly show the potential for slowdowns of 10, 100, or even 1000 when locality of reference is poor.

The lower graph in Figure 2 plots the factor by which the protected methods use more processing time exclusive of disk delays. Again we see interesting trends. In the case of VM protection the workload ratio is

$$\begin{aligned} & \frac{n_{ir}t_{inst} + (\bar{p}_c + p_{dc})t_{line} + (\bar{p}_m + p_{dm})n_{page}t_{aes}}{n_{ir}t_{inst} + (\bar{p}_c + p_{dc})t_{line}} \\ = \quad & 1 + \left(\frac{\bar{p}_m + p_{cm}}{\bar{p}_c + p_{dc}}\right)\frac{n_{page}t_{aes}}{t_{line}}. \end{aligned}$$

This explains the observed behavior : when the cache miss rate ($\bar{p}_c$) is very small the ratio is dominated by some multiple of $n_{page}t_{aes}/t_{line} = 571.4n_{page}$. That term diminishes as the cache miss rate increases, approaching unity. Something similar occurs in the case of PM protection. Here the ratio is

$$\frac{n_{ir}t_{inst} + (\bar{p}_s + p_{ds})(n_{page}t_{aes} + t_{trans})}{n_{ir}t_{inst} + (\bar{p}_c + p_{dc})t_{line}}.$$

Assuming that $\bar{p}_s$ and $\bar{p}_c$ increase together, the ratio goes from being dominated by the $n_{ir}t_{hit}$ terms at very low miss rates, to increase asymptotically to $571.4n_{page}$. It is important to note that $571.4n_{page}$ must serve as an upper bound on the execution time ratio, for that ratio is maximized when the cost associated with the disk is minimized.

The possibility of a slowdown on the order of 1000 is daunting. However, high workload ratios need not always be problematic, when the workload increase is dominated by the delay due to disk activity. The flat-line behavior of the VM execution ratio is an example of this—observe that under low cache miss rates the PM workload ratio is as high as 100. A similar principal applies when the protected code runs in a multiprogramming environment. This could be expected if a physically secured host uses specialized hardware and

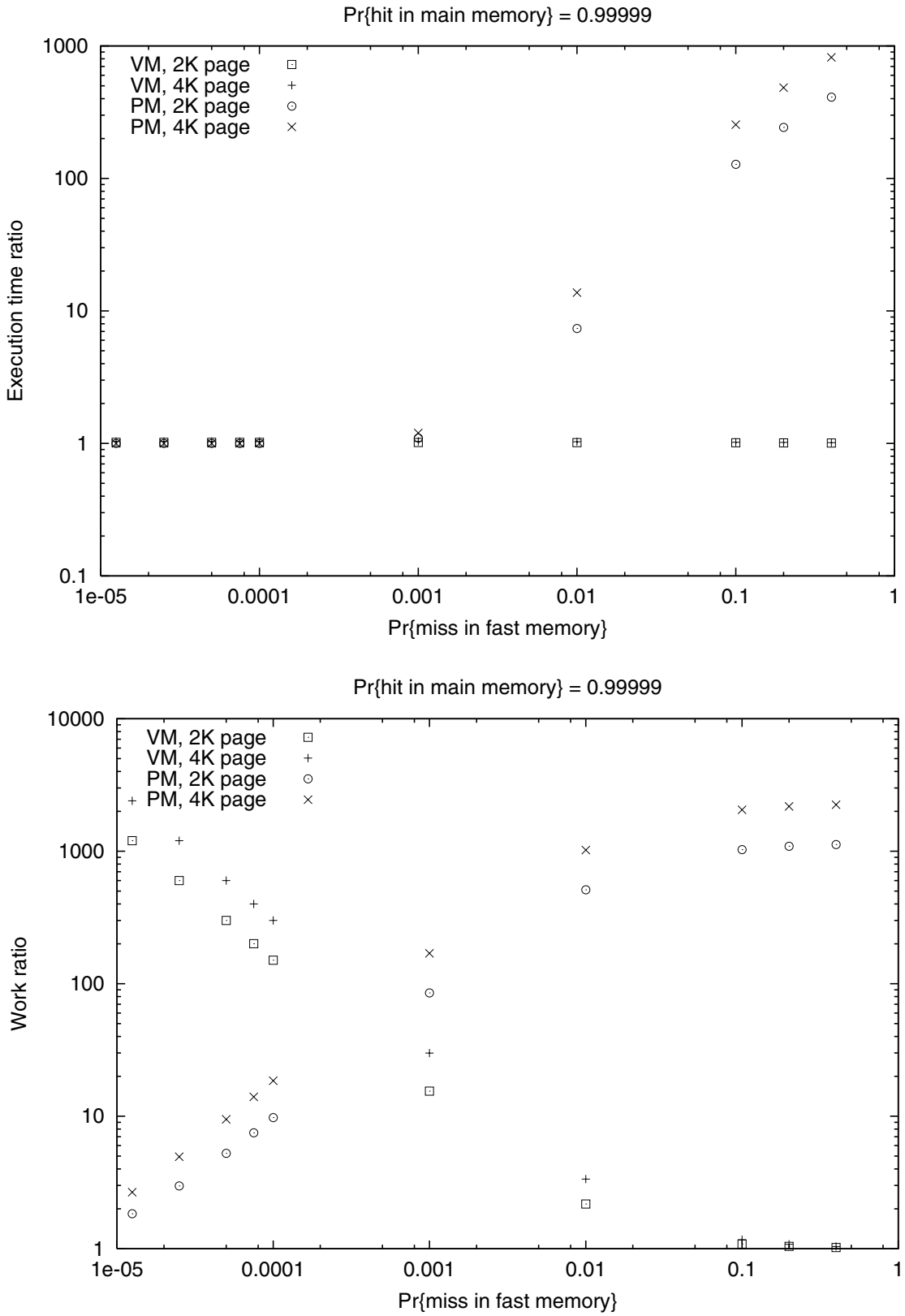

Figure 2: The top graph shows overall execution time slowdown due to cryptographic measures, as a function of the miss rate in the fast memory, the bottom graph depicts the factor by which time spent in non-disk computation increases due to cryptographic measures. VM denotes protection of program on disk, PM denotes protection of program in main memory. Ratios are computed separately under the assumption of 2K and 4K byte memory pages. Both graphs assume a main memory miss rate of 10^{-5}.

Table 2: Characteristics of Lucas and Sixtrack Benchmark Simulations

(a) Hit Rates into Main Memory, Cache, and Scratchpad

trace	p_m	p_c	p_s
lucas 2k	0.99998052	0.99964415	0.99998052
lucas 4k	0.99999023	0.99964415	0.99999023
sixtrack 2k	0.99998996	0.99624699	0.99996118
sixtrack 4k	0.99999494	0.99624699	0.99997313

(b) Per-instruction Costs (in nanoseconds) from Lucas and Sixtrack Benchmarks.

trace	cpu cost	encryption cost	disk cost	normal execution	workload	protected execution
lucas 2k (VM)	1.24	1.84	121.20	122.44	3.08	124.28
lucas 2k (PM)	1.22	2.77	121.13	122.35	3.99	125.11
lucas 4k (VM)	1.24	1.84	60.61	61.86	3.09	63.70
lucas 4k (PM)	1.22	2.78	60.70	61.92	4.00	64.69
sixtrack 2k (VM)	1.24	1.84	121.32	122.56	3.08	124.40
sixtrack 2k (PM)	1.22	2.77	121.12	122.34	3.99	125.11
sixtrack 4k (VM)	1.24	1.84	61.72	62.97	3.09	64.81
sixtrack 4k (PM)	1.22	2.78	61.66	62.88	4.00	65.66

software; potentially many secured applications might be running concurrently. In this context an individual program's execution time is much less important than the system's throughput. In a multiprogrammed environment a secured application taking a page fault will have to wait for the disk to serve the disk requests of a number of other applications first. From the point of view of the application it is as though the disk was simply much slower than if the host were dedicated to the application. Thus the presence of those applications effectively increases the time spent waiting for the disk. Naturally this increases the execution time, but it can also serve to cause the larger disk wait to dominate the execution time in a way it would not if the host were dedicated to the application. The execution time ratio (adjusted now to reflect increased disk delay under the unprotected case) may once again approach unity.

7 CONCLUSIONS

This paper uses analytic modeling and discrete-event simulation to evaluate the performance impact of securing application programs from unauthorized observation or execution. It studies two methods based on encrypting the application executable and its data. One method allows the executable to exist in plaintext in main memory, but makes sure that any representation of any part of it on the disk is encrypted. The second method hardens the application further, ensuring that no plaintext representation of it ever exists in main memory.

We find that the costs of the first approach are always dominated by the inherent cost of virtual memory, and so it suffers no relative performance impact. The second method is seen to not impact performance when the protected program has good cache locality. Our fundamental conclusion is that while the amount of additional computation needed to support protection may be large, in a variety of common circumstances those costs are dominated by inherent disk costs, so that the *relative* impact of the extra computation is not large.

8 ACKNOWLEDGMENTS

This research was supported under Award number 2000-DT-CX-K001 from the U.S. Department of Homeland Security, Science and Technology Directorate. Points of view in this document are those of the author(s) and do not necessarily represent the official position of the U.S. Department of Homeland Security or the Science and Technology Directorate. In addition this research was supported by SPAWAR contract N66001-04-C-6013, and NSF Grant CCR-0209144. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

REFERENCES

Avissar, O., R. Barua, and D. Stewart. 2002. An optimal memory allocation scheme for scratch-pad-based embedded systems. *Trans. on Embedded Computing Sys.* 1 (1): 6–26.

Banakar, R., S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. 2002. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, 73–78. New York, NY, USA: ACM Press.

Clark, P. 2003, January. Project targets military security. *Military Information Technology (On-*

Line Edition) 7 (1). see www.military-information-technology.com/article.cfm?DocID=38.

Collberg, C., and C. Thomborson. 2002, August. Watermarking, tamper-proofing, and obfuscation tools for software protection. *IEEE Transactions on Software Engineering* 28 (8): 735–746.

Daemen, J., and V. Rijmen. 2002. *The design of rijndael*. Springer.

Griffiths, A. L. 2005. Binary protection schemes. *Code Breaker's Journal* 2 (1). see www.codebreakers-journal.com/viewissue.php?id=3.

grugq, and scut. 2001. Armouring the elf : Binary encryption on the unix platform. see www.phrack.org/phrack/58/p58-0x05.

Jaeschke, R. 1990, June. Miscellanea. *The Journal of C Language Translation* 2 (1): 71–80.

Lee, E., and R. Katz. 1993, May. An analytic performance model for disk arrays. In *Proceedings of the 1994 ACM SIGMETRICS Conference*, 98–109.

Mehta, N., and S. Clowes. 2003. see www.securiteam.com/tools/5XP041FA0U.html.

Naumovich, G., and N. Memon. 2003, July. Preventing piracy, reverse engineering, and tampering. *Computer* 36 (7): 64–71.

Ravindran, R. A., P. D. Nagarkar, G. S. Dasika, E. D. Marsman, R. M. Senger, S. A. Mahlke, and R. B. Brown. 2005. Compiler managed dynamic instruction placement in a low-power code cache. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, 179–190. Washington, DC, USA: IEEE Computer Society.

SPEC 2000. Cfp2000 (floating point component of spec cpu2000. see www.spec.org/cpu2000/CFP2000.

Stytz, M., and J. Whittaker. 2003, January. Software protection: Security's last stand? *IEEE Security and Privacy* 1 (1): 95–98.

AUTHOR BIOGRAPHIES

DAVID M. NICOL is Professor of Electrical and Computer Engineering at the University of Illinois, Urbana-Champaign, and member of the Coordinated Sciences Laboratory. He is co-author of the textbook *Discrete-Event Systems Simulation*, and served as Editor-in-Chief at ACM TOMACS from 1997-2003. He was the General Chair of the 2005 Conference on Principles of Advanced and Distributed Simulation, and the General Chair of the 2006 Winter Simulation Conference. From 1996-2003 he served as the Editor-in-Chief of the ACM *Transactions on Modeling and Computer Simulation*. He has a B.A. in mathematics from Carleton College (1979), an M.S. (1983) and Ph.D. (1985) in computer science from the University of Virginia. His research interests are in high performance computing, performance analysis, simulation and modeling, and network security. He is a Fellow of the IEEE.

HAMED OKHRAVI is a graduate student in the department of Electrical and Computer Engineering at the University of Illinois, Urbana-Champaign. He earned a B.Sc. degree in Electrical Engineering in 2003 at the Sharif University of Technology, Tehran, Iran. His research interests are in computer security and cryptography.