Randomized Instruction Set Emulation

ELENA GABRIELA BARRANTES, DAVID H. ACKLEY, STEPHANIE FORREST, and DARKO STEFANOVIĆ
University of New Mexico

Injecting binary code into a running program is a common form of attack. Most defenses employ a "guard the doors" approach, blocking known mechanisms of code injection. *Randomized instruction set emulation* (RISE) is a complementary method of defense, one that performs a hidden randomization of an application's machine code. If foreign binary code is injected into a program running under RISE, it will not be executable because it will not know the proper randomization. The paper describes and analyzes RISE, describing a proof-of-concept implementation built on the open-source Valgrind IA32-to-IA32 translator. The prototype effectively disrupts binary code injection attacks, without requiring recompilation, linking, or access to application source code. Under RISE, injected code (attacks) essentially executes random code sequences. Empirical studies and a theoretical model are reported which treat the effects of executing random code on two different architectures (IA32 and PowerPC). The paper discusses possible extensions and applications of the RISE technique in other contexts.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*; D.3.4 [**Programming Languages**]: Processors—*Interpreters, runtime environments*

General Terms: Security

Additional Key Words and Phrases: Automated diversity, randomized instruction sets, software diversity

An earlier version of this paper was published as Barrantes, E. G., Ackley, D. H., Forrest, S., Palmer, T. S., Stefanović, D., and Dai Zovi, D. 2003. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pp. 281–289. This version adds a detailed model and analysis of the safety of random bit execution, and presents additional empirical results on the prototype's effectiveness and performance.

The authors gratefully acknowledge the partial support of the National Science Foundation (grants ANIR-9986555, CCR-0219587, CCR-0085792, CCR-0311686, EIA-0218262, EIA-0238027, and EIA-0324845), the Office of Naval Research (grant N00014-99-1-0417), Defense Advanced Research Projects Agency (grants AGR F30602-00-2-0584 and F30602-02-1-0146), Sandia National Laboratories, Hewlett-Packard gift 88425.1, Microsoft Research, and Intel Corporation. Any opinions, findings, conclusions, or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

Authors' address: Department of Computer Science, University of New Mexico, MSC01 1130, Albuquerque, NM 87131-1386.

Stephanie Forrest is also with the Santa Fe Institute, 1399 Hyde Park Rd, Santa Fe, NM 87501.

1. INTRODUCTION

Standardized machine instruction sets provide consistent interfaces between software and hardware, but they are a double-edged sword. Although they yield great productivity gains by enabling independent development of hardware and software, the ubiquity of well-known instructions sets also allows a single attack designed around an exploitable software flaw to gain control of thousands or millions of systems. Such attacks could be stopped or greatly hindered if each protected system could be economically destandardized, so that a different attack would have to be created specifically for each new target, using information that was difficult or impossible for an outsider to obtain. The automatic diversification we explore in this paper is one such destandardization technique.

Many existing defenses against machine code injection attacks block the known routes by which foreign code is placed into a program's execution path. For example, stack defense mechanisms [Chiueh and Hsu 2001; Cowan et al. 1998; Etoh and Yoda 2000, 2001; Forrest et al. 1997; Frantzen and Shuey 2001; Nebenzahl and Wool 2004; Prasad and Chiueh 2003; Vendicator 2000; Xu et al. 2002] protect return addresses and defeat large classes of buffer overflow attacks. Other mechanisms defend against buffer overflows elsewhere in program address space [PaX Team 2003], against alternative overwriting methods [Cowan et al. 2001], or guard from known vulnerabilities through shared interfaces [Avijit et al. 2004; Baratloo et al. 2000; Lhee and Chapin 2002; Tsai and Singh 2001]. Our approach is functionally similar to the PAGEEXEC feature of PaX [PaX Team 2003], an issue we discuss in Section 6.

Rather than focusing on any particular code injection pathway, a complementary approach would disrupt the operation of the injected code itself. In this paper we describe *randomized instruction set emulation* (RISE), which uses a machine emulator to produce automatically diversified instruction sets. With such instruction set diversification, each protected program has a different and secret instruction set, so that even if a foreign attack code manages to enter the execution stream, with very high probability the injected code will fail to execute properly.

In general, if there are many possible instruction sets compared to the number of protected systems and the chosen instruction set in each case is externally unobservable, different attacks must be crafted for each protected system and the cost of developing attacks is greatly increased. In RISE, each byte of protected program code is scrambled using pseudorandom numbers seeded with a random key that is unique to each program execution. Using the scrambling constants it is trivial to recover normal instructions executable on the physical machine, but without the key it is infeasible to produce even a short code sequence that implements any given behavior. Foreign binary code that manages to reach the emulated execution path will be descrambled without ever having been correctly scrambled, foiling the attack, and producing pseudorandom code that will usually crash the protected program.

1.1 Threat Model

The set of attacks that RISE can handle is slightly different from that of many defense mechanisms, so it is important to identify the RISE threat model clearly.

Our specific threat model is *binary code injection from the network into an executing program*. This includes many real-world attack mechanisms, but explicitly excludes several others, including the category of attacks loosely grouped under the name "return into libc" [Nergal 2001] which modify data and addresses so that code already existing in the program is subverted to execute the attack. These attacks might or might not use code injection as part of the attack. Most defenses against code injection perform poorly against this category as it operates at a different level of abstraction; complementary defense techniques are needed, and have been proposed, such as address obfuscation [Bhatkar et al. 2003; Chew and Song 2002; PaX Team 2003], which hide and/or randomize existing code locations or interface access points.

The restriction to code injection attacks excludes data only attacks such as nonhybrid versions of the "return into libc" class mentioned above, while focusing on binary code excludes attacks such as macro viruses that inject code written in a higher-level language. Finally, we consider only attacks that arrive via network communications and therefore we treat the contents of local disks as trustworthy before an attack has occurred.

In exchange for these limitations, RISE protects against all binary code injection attacks, regardless of the method by which the machine code is injected. By defending the code itself, rather than any particular access route into the code, RISE offers the potential of blocking attacks based on injection mechanisms that have yet to be discovered or revealed.

This threat model is related to, but distinct from, other models used to characterize buffer overflow attacks [Cowan et al. 2000, 2001]. It includes any attack in which native code is injected into a running binary, even by means that are not obviously buffer overflows, such as misallocated malloc headers, footer tags [Security Focus 2003; Xu et al. 2003], and format string attacks that write a byte to arbitrary memory locations [Gera and Riq 2002; Newsham 2000]. RISE protects against injected code arriving by any of these methods. On the other hand, other defense mechanisms, such as the address obfuscation mentioned above, can prevent attacks that are specifically excluded from our code injection threat model.

We envision the relatively general code-based mechanism of RISE being used in conjunction with data and address diversification-based mechanisms to provide deeper, more principled, and more robust defenses against both known and unknown attacks.

1.2 Overview

This paper describes a proof-of-concept RISE system, which builds randomized instruction set support into a version of the Valgrind IA32-to-IA32 binary translator [Nethercote and Seward 2003; Seward and Nethercote 2004]. Section 2 describes a randomizing loader for Valgrind that scrambles code sequences loaded into emulator memory from the local disk using a hidden random key. Then, during Valgrind's emulated instruction fetch cycle, fetched instructions are unscrambled, yielding the unaltered IA32 machine code sequences of the protected application. The RISE design makes few demands on the supporting

emulator and could be easily ported to any binary-to-binary translator for which source code is available.

Section 3 reports empirical tests of the prototype and confirms that RISE successfully disrupts a range of actual code injection attacks against otherwise vulnerable applications. In addition, it highlights the extreme fragility of typical attacks and comments on performance issues.

A basic property of the RISE defense mechanism is that if an attack manages to inject code by any means, essentially random machine instructions will be executed. Section 4 investigates the likely effects of such an execution in several different execution contexts. Experimental results are reported and theoretical analyses are given for two different architectures. There is always a possibility that random bits could create valid instructions and instruction sequences. We present empirical data suggesting that the majority of random code sequences will produce an address fault or illegal instruction quickly, causing the program to abort. Most of the remaining cases throw the program into a loop, effectively stopping the attack. Either way, an attempted takeover is downgraded into a denial-of-service attack against the exploitable program.

Unlike compiled binary code, which uses only a well-defined and often relatively small selection of instructions, random code is unconstrained. The behavior of random code execution in the IA32 architecture can involve the effects of undocumented instructions and whatever instruction set extensions (e.g., MMX, SSE, and SSE2) are present, as well as the effects of random branch offsets combined with multibyte, variable-length instructions. Although those characteristics complicate a tight theoretical analysis of random bit executions on the IA32, models for more constrained instruction set architectures, such as the PowerPC, lead to a closer fit to the observed data.

Section 6 summarizes related work, Section 7 discusses some of the implications and potential vulnerabilities of the RISE approach, and Section 8 concludes the paper.

2. TECHNICAL APPROACH AND IMPLEMENTATION

This section describes the prototype implementation of RISE using Valgrind [Nethercote and Seward 2003; Seward and Nethercote 2004] for the Intel IA32 architecture. Our strategy is to provide each program copy its own unique and private instruction set. To do this, we consider what is the most appropriate machine abstraction level, how to scramble and descramble instructions, when to apply the randomization and when to descramble, and how to protect interpreter data. We also describe idiosyncrasies of Valgrind that affected the implementation.

2.1 Machine Abstraction Level

The native instruction set of a machine is a promising computational level for automated diversification because all computer functionality can be expressed in machine code. This makes the machine-code level desirable to attack and protect. However, automated diversification is feasible at higher levels of abstraction, although there are important constraints on suitable candidates.

Language diversification seems most promising for languages that are interpreted or executed directly by a virtual machine. Randomizing source code for a compiled language would protect only against injections at compile time. An additional constraint is the possibility of crafting attacks at the selected language level. Although it is difficult to evaluate this criterion in the abstract, we could simply choose languages for which those attacks have already been shown to exist, such as Java, Perl, and SQL [Harper 2002]. And in fact, proposals for diversifying these higher levels have been made [Boyd and Keromytis 2004; Kc et al. 2003]. Macro languages provide another example of a level that could be diversified to defeat macro viruses.

Finally, it is necessary to have a clear trust boundary between internal and external programs so that it is easy to decide which programs should be randomized. The majority of programs should be internal to the trust boundary, or the overhead of deciding what is trusted and untrusted will become too high. This requirement eliminates most web-client scripting languages such as Javascript because a user decision about trust would be needed every time a Javascript program was going to be executed on a client. A native instruction set, with a network-based threat model, provides a clear trust boundary, as all legitimately executing machine code is stored on a local disk.

An obvious drawback of native instruction sets is that they are traditionally physically encoded and not readily modifiable. RISE therefore operates at an intermediate level, using software that performs binary-to-binary code translation. The performance impact of such tools can be minimal [Bala et al. 2000; Bruening et al. 2001]. Indeed, binary-to-binary translators sometimes improve performance compared to running the programs directly on the native hardware [Bala et al. 2000].

For ease of research and dissemination, we selected the open-source emulator, Valgrind, for our prototype. Although Valgrind is described primarily as a tool for detecting memory leaks and other program errors, it contains a complete IA32-to-IA32 binary translator. The primary drawback of Valgrind is that it is very slow, largely owing to its approach of translating the IA32 code into an intermediate representation and its extensive error checking. However, the additional slowdown imposed by adding RISE to Valgrind is modest, and we are optimistic that porting RISE to a more performance-oriented emulator would yield a fully practical code defense.

2.2 Instruction Set Randomization

Instruction set randomization could be as radical as developing a new set of opcodes, instruction layouts, and a key-based toolchain capable of generating the randomized binary code. And, it could take place at many points in the compilation-to-execution spectrum. Although performing randomization early could help distinguish code from data, it would require a full compilation environment on every machine, and recompiled randomized programs would likely have one fixed key indefinitely. RISE randomizes as late as possible in the process, scrambling each byte of the trusted code as it is loaded into the emulator, and then unscrambling it before execution. Deferring the randomization to

load time makes it possible to scramble and load existing files in the executable and linking format (ELF) [Tool Interface Standards Committee . 1995] directly, without recompilation or source code, provided we can reliably distinguish code from data in the ELF file format.

The unscrambling process needs to be fast, and the scrambling process must be as hard as possible for an outsider to deduce. Our current default approach is to generate at load time a pseudorandom sequence the length of the overall program text using the Linux `/dev/urandom` device [Tso 1998], which uses a secret pool of true randomness to seed a pseudorandom stream generated by feedback through SHA1 hashing. The resulting bytes are simply XORed with the instruction bytes to scramble and unscramble them. In addition, it is possible to specify the length of the key, and a smaller key can be tiled over the process code. If the underlying truly random key is long enough, and as long as it is infeasible to invert SHA1 [Schneier 1996], we can be confident that an attacker cannot break the entire sequence. The security of this encoding is discussed further in Section 7.

2.3 Design Decisions

Two important aspects of the RISE implementation are how it handles shared libraries and how it protects the plaintext executable.

Much of the code executed by modern programs resides in shared libraries. This form of code sharing can significantly reduce the effect of the diversification, as processes must use the same instruction set as the libraries they require. When our load-time randomization mechanism writes to memory that belongs to shared objects, the operating system does a copy-on-write, and a private copy of the scrambled code is stored in the virtual memory of the process. This significantly increases memory requirements, but increases interprocess diversity and avoids having the plaintext code mapped in the protected processes' memory. This is strictly a design decision, however. If the designer is willing to sacrifice some security, it can be arranged that processes using RISE share library keys, and so library duplication could be avoided.

Protecting the plaintext instructions inside Valgrind is a second concern. As Valgrind simulates the operation of the CPU, during the fetch cycle when the next byte(s) are read from program memory, RISE intercepts the bytes and unscrambles them; the scrambled code in memory is never modified. Eventually, however, a plaintext piece of the program (semantically equivalent to the block of code just read) is written to Valgrind's cache. From a security point of view, it would be best to separate the RISE address space completely from the protected program address space, so that the plaintext is inaccessible from the program, but as a practical matter this would slow down emulator data accesses to an extreme and unacceptable degree. For efficiency, the interpreter is best located in the same address space as the target binary, but of course this introduces some security concerns. A RISE-aware attacker could aim to inject code into a RISE data area, rather than that of the vulnerable program. This is a problem because the cache cannot be encrypted. To protect the cache its pages are kept as read-and-execute only. When a new translated basic block is ready to be

written to the cache, we mark the affected pages as writable, execute the write action, and restore the pages to their original nonwritable permissions. A more principled solution would be to randomize the location of the cache and the fragments inside it, a possibility for future implementations of RISE.

2.4 Implementation Issues

Our current implementation does not handle self-modifying code, but it has a primitive implementation of an interface to support dynamically generated code. We consider arbitrary self-modifying code as an undesirable programming practice and agree with Valgrind's model of not allowing it. However, it is desirable to support legitimate dynamically generated code, and eventually we intend to provide a complete interface for this purpose.

An emulator needs to create a clear boundary between itself and the process to be emulated. In particular, the emulator should not use the same shared libraries as the process being emulated. Valgrind deals with this issue by adding its own implementation of all library functions it uses, with a local modified name for example, `VGplain_printf` instead of `printf`. However, we discovered that Valgrind occasionally jumped into the target binary to execute low-level functions (e.g., `__umoddi` and `__udivdi`). When that happened, the processor attempted to execute instructions that had been scrambled for the emulated process, causing Valgrind to abort. Although this was irritating, it did demonstrate the robustness of the RISE approach in that these latent boundary crossings were immediately detected. We worked around these dangling unresolved references by adding more local functions to Valgrind and renaming affected symbols with local names (e.g., `rise_umoddi` instead of "%" (the modulo operator)).

A more subtle problem arises because the IA32 does not impose any data and code separation requirement, and some compilers insert dispatch tables directly in the code. In those cases, the addresses in such internal tables are scrambled at load time (because they are in a code section), but are not descrambled at execution time because they are read as data. Although this does not cause an illegal operation, it causes the emulated code to jump to a random address and fail inappropriately. At interpretation time, RISE looks for code sequences that are typical for jump-table referencing and adds machine code to check for in-code references into the block written to the cache. If an in-code reference is detected when the block is executing, our instrumentation descrambles the data that was retrieved and passes it in the clear to the next (real) instruction in the block. This scheme could be extended to deal with the general case of using code as data by instrumenting every dereference to check for in-code references. However, this would be computationally expensive, so we have not implemented it in the current prototype. Code is rarely used as data in legitimate programs except in the case of virtual machines, which we address separately.

An additional difficulty was discovered with Valgrind itself. The thread support implementation and the memory inspection capabilities require Valgrind to emulate itself at certain moments. To avoid infinite emulation regress, it has a special workaround in its code to execute some of its own functions natively during this self-emulation. We handled this by detecting Valgrind's own address

ranges and treating them as special cases. This issue is specific to Valgrind, and we expect not to encounter it in other emulators.

3. EFFICACY AND PERFORMANCE OF RISE

The results reported in this section were obtained using the RISE prototype, available under the GPL from `http://cs.unm.edu/~immsec`. We have tested RISE's ability to run programs successfully under normal conditions and its ability to disrupt a variety of machine code injection attacks. The attack set contained 20 synthetic and 15 real attacks.

The synthetic attacks were obtained from two sources. Two attacks, published by Fayolle and Glaume [2002], create a vulnerable buffer—in one case on the heap and in the other case on the stack—and inject shellcode into it. The remaining 18 attacks were executed with the attack toolkit provided by Wilander and Kamkar and correspond to their classification of possible buffer overflow attacks [Wilander and Kamkar 2003] according to technique (direct or pointer redirection), type of location (stack, heap, BSS, or data segment), and attack target (return address, old base pointer, function pointer, and longjump buffer). Without RISE, either directly on the processor or using Valgrind, all of these attacks successfully spawn a shell. Using RISE, the attacks are stopped.

The real attacks were launched from the CORE impact attack toolkit [CORE Security 2004]. We selected 15 attacks that satisfied the following requirements of our threat model and the chosen emulator: the attack is launched from a remote site; the attack injects binary code at some point in its execution; and, the attack succeeds on a Linux OS. Because Valgrind runs under Linux; we focused on Linux distributions, reporting data from Mandrake 7.2 and versions of RedHat from 6.2 to 9.

3.1 Results

All real (nonsynthetic) attacks were tested on the vulnerable applications before retesting with RISE. All of them were successful against the vulnerable services without RISE, and they were all defeated by RISE (Table I).

Based on the advisories issued by CERT in the period between 1999 and 2003, Xu et al. [2003] classify vulnerabilities that can inject binary code into a running process according to the method used to modify the execution flow: buffer overflows, format string vulnerabilities, malloc/free, and integer manipulation errors. Additionally, the injected code can be placed in different sections of the process (stack, heap, data, BSS). The main value of RISE is its imperviousness to the entry method and/or location of the attack code, as long as the attack itself is expressed as binary code. This is illustrated by the diversity of vulnerability types and shellcode locations used in the real attacks (columns 3 and 4 of Table I).

The available synthetic attacks are less diverse in terms of vulnerability type. They are all buffer overflows. However, they do have attack code location variety (stack, heap, and data), and more importantly, they have controlled diversity of corrupted code address types (return address, old base pointer, function pointer, and longjump buffer as either local variable or parameter),

Table I. Results of Attacks Against Real Applications Executed under RISE

Attack	Linux Distribution	Vulnerability	Location of Injected Code	Stopped by RISE
Apache OpenSSL SSLv2	RedHat 7.0 & 7.2	Buffer overflow and malloc/free	Heap	√
Apache mod php	RedHat 7.2	Buffer overflow	Heap	√
Bind NXT	RedHat 6.2	Buffer overflow	Stack	√
Bind TSIG	RedHat 6.2	Buffer overflow	Stack	√
CVS flag insertion heap exploit	RedHat 7.2 & 7.3	Malloc/free	Heap	√
CVS pserver double free	RedHat 7.3	Malloc/Free	Heap	√
PoPToP Negative Read	RedHat 9	Integer error	Heap	√
ProFTPD _xlate_ascii _write off-by-two	RedHat 9	Buffer overflow	Heap	√
rpc.statd format string	RedHat 6.2	Format string	GOT	√
SAMBA nttrans	RedHat 7.2	Buffer overflow	Heap	√
SAMBA trans2	RedHat 7.2	Buffer overflow	Stack	√
SSH integer overflow	Mandrake 7.2	Integer error	Stack	√
sendmail crackaddr	RedHat 7.3	Buffer overflow	Heap	√
wuftpd format string	RedHat 6.2–7.3	Format string	Stack	√
wuftpd glob "~{"	RedHat 6.2–7.3	Buffer overflow	Heap	√

Column 1 gives the exploit name (and implicitly the service against which it was targeted).
The vulnerability type and attack code (shellcode) locations are included (columns 3 and 4, respectively).
The result of the attack is given in column 5.

and offer either direct or indirect execution flow hijacking (see Wilander and Kamkar [2003]). All of Wilander's attacks have the shellcode located in the data section. Both of Fayolle and Glaume's exploits use direct return address pointer corruption. The stack overflow injects the shellcode on the stack, and the heap overflow locates the attack code on the heap. All synthetic attacks are successful (spawn a shell) when running natively on the processor or over unmodified Valgrind. All of them are stopped by RISE (column 5 of Table II).

When we originally tested real attacks and analyzed the logs generated by RISE, we were surprised to find that nine of them failed without ever executing the injected attack code. Further examination revealed that this was due to various issues with Valgrind itself, which have been remedied in later versions. The current RISE implementation in Valgrind 2.0.0 does not have this behavior. All attacks (real and synthetic) are able to succeed when the attacked program runs over Valgrind, just as they do when running natively on the processor.

These results confirm that we successfully implemented RISE and that a randomized instruction set prevents injected machine code from executing, without the need for any knowledge about how or where the code was inserted in process space.

3.2 Performance

Being emulation based, RISE introduces execution costs that affect application performance. For a proof-of-concept prototype, correctness and defensive power were our primary concerns, rather than minimizing resource overhead. In this section, we describe the principal performance costs of the RISE approach,

Table II. Results of the Execution of Synthetic Attacks under RISE

Type of Overflow	Shellcode Location	Exploit Origin	Number of ≠ Pointer Types	Stopped by RISE
Stack direct	Data	Wilander and Kamkar [2003]	6	6 (100%)
Data direct	Data	Wilander and Kamkar [2003]	2	2 (100%)
Stack indirect	Data	Wilander and Kamkar [2003]	6	6 (100%)
Data indirect	Data	Wilander and Kamkar [2003]	4	4 (100%)
Stack direct	Stack	Fayolle and Glaume [2002]	1	1 (100%)
Stack direct	Heap	Fayolle and Glaume [2002]	1	1 (100%)

Type of overflow (column 1) denotes the location of the overflowed buffer (stack, heap or data) and the type of corruption executed: *direct* modifies a code pointer during the overflow (such as the return address), and *indirect* modifies a data pointer that eventually is used to modify a code pointer.
Shellcode location (column 2) indicates the segment where the actual malicious code was stored.
Exploit origin (column 3) gives the paper from which the attacks were taken.
The number of pointer types (column 4) defines the number of different attacks that were tried by varying the type of pointer that was overflowed.
Column 5 gives the number of different attacks in each class that were stopped by RISE.

which include a once-only time cost for code randomization during loading, time for derandomization while the process executes, and space overheads.

Although in the following we assume an all-software implementation, RISE could also be implemented with hardware support, in which case we would expect much better performance because the coding and decoding could be performed directly in registers rather than executing two different memory accesses for each fetch.

The size of each RISE-protected process is increased because it must have its own copy of any library it uses. Moreover, the larger size is as much as doubled to provide space for the randomization mask.[1]

A software RISE uses dynamic binary translation, and pays a runtime penalty for this translation. Valgrind amortizes interpretation cost by storing translations in a cache, which allows native-speed execution of previously interpreted blocks.

Valgrind is much slower than binary translators [Bala et al. 2000; Bruening et al. 2001] because it converts the IA32 instruction stream into an intermediate representation before creating the code fragment. However, we will give some evidence that long-running, server-class processes can execute at reasonable speeds, and these are precisely the ones for which RISE is most needed.

As an example of this effect, Table III provides one data point about the long-term runtime costs of using RISE, using the Apache web server in the face of a variety of nonattack workloads. Classes 0 to 3, as defined by SPEC Inc. [1999], refer to the size of the files that are used in the workload mix. Class 0 is the least I/O intensive (files are less than 1 KB long), and class 3 is the one that uses the most I/O (files up to 1000 KB long). As expected, on I/O bound mixes, the throughput of Apache running over RISE is closer to Apache running

[1]A RISE command-line switch controls the length of the mask, which is then tiled to cover the program. A 1000-byte mask, for example, would be a negligible cost for mask space, and very probably would provide adequate defense. In principle, however, it might open a within-run vulnerability owing to key reuse.

Table III. Comparison of the Average Time Per Operation between Native Execution of Apache and Apache over RISE

	Native Execution		Execution over RISE		RISE/
Mix Type	Mean (ms)	Std. Dev.	Mean (ms)	Std. Dev.	Native
Class 0	177.32	422.22	511.73	1,067.79	2.88
Class 1	308.76	482.31	597.11	1,047.23	1.93
Class 2	1,230.75	624.58	1,535.24	1,173.57	1.25
Class 3	10,517.26	3,966.24	11,015.74	4,380.26	1.05
Total	493.80	1,233.56	802.63	1,581.50	1.62

Presented times were obtained from the second iteration in a standard SPECweb99 configuration (300 s warm up and 1200 s execution).

directly on the processor.[2] Table III shows that the RISE prototype slows down by a factor of no more than 3, and sometimes by as little as 5%, compared with native execution, as observed by the client. These results should not be taken as a characterization of RISE's performance, but as evidence that cache-driven amortization and large I/O and network overheads make the CPU performance hit of emulation just one (and possibly not the main) factor in evaluating the performance of this scheme.

By contrast, short interactive jobs are more challenging for RISE performance, as there is little time to amortize mask generation and cache filling. For example, we measured a slowdown factor of about 16 end-to-end when RISE protecting all the processes invoked to make this paper from LaTeX source.

Results of the Dynamo project suggest that a custom-built dynamic binary translator can have much lower overheads than Valgrind, suggesting that a commercial-grade RISE would be fast enough for widespread use; in long-running contexts where performance is less critical, even our proof-of-concept prototype might be practical.

4. RISE SAFETY: EXPERIMENTS

Code diversification techniques such as RISE rely on the assumption that random bytes of code are unlikely to execute successfully. When binary code is injected by an attacker and executes, it is first derandomized by RISE. Because the attack code was never prerandomized, the effect of derandomizing is to transform the attack code into a random byte string. This is invisible to the interpretation engine, which will attempt to translate, and possibly execute, the string. If the code executes at all, it clearly will not have the effect intended by the attacker. However, there is some chance that the random bytes might correspond to an executable sequence, and an even smaller chance that the executed sequence of random bytes could cause damage. In this section, we measure the likelihood of these events under several different assumptions, and in the following section we develop theoretical estimates.

Our approach is to identify the possible actions that randomly formed instructions in a sequence could perform and then to calculate the probabilities

[2]The large standard deviations are typical of SPECweb99, as web server benchmarks have to model long-tailed distributions of request sizes [Nahum 2002; SPEC Inc. 1999].

for these different events. There are several broad classes of events that we consider: illegal instructions that lead to an error signal, valid execution sequences that lead to an infinite loop or a branch into valid code, and other kinds of errors. There are several subtle complications involved in the calculations, and in some cases we make simplifying assumptions. The simplifications lead to a conservative estimate of the risk of executing random byte sequences.

4.1 Possible Behaviors of Random Byte Sequences

First, we characterize the possible events associated with a generic processor or emulator attempting to execute a random symbol. We use the term *symbol* to refer to a potential execution unit, because a symbol's length in bytes varies across different architectures. For example, instruction length in the PowerPC architecture is exactly 4 bytes and in the IA32 it can vary between 1 and 17 bytes. Thus, we adopt the following definitions:

(1) A *symbol* is a string of l bytes, which may or may not belong to the instruction set. In a RISC architecture, the string will always be of the same length, while for CISC it will be of variable length.
(2) An *instruction* is a symbol that belongs to the instruction set.

In RISE there is no explicit recognition of an attack, and success is measured by how quickly and safely the attacked process is terminated. Process termination occurs when an error condition is generated by the execution of random symbols. Thus, we are interested in the following questions:

(1) How soon will the process *crash* after it begins executing random symbols? (Ideally, in the first symbol.)
(2) What is the probability that an execution of random bytes will branch to valid code or enter an infinite loop (*escape*)? (Ideally, 0.)

Figure 1 illustrates the possible outcomes of executing a single random symbol. There are three classes of outcome: an error that generates a signal, a branch into executable memory in the process space that does not terminate in an error signal (which we call *escape*), and the simple execution of the symbol with the program pointer moving to the next symbol in the sequence. Graph traversal always begins in the *start* state, and proceeds until a terminating node is reached (*memory error signal*, *instruction-specific error signal*, *escape*, or *start*).

The term *crash* refers to any error signal (the states labeled *invalid opcode*, *specific error signal*, and *memory error signal* in Figure 1). Error signals do not necessarily cause process termination due to error, because the process could have defined handlers for some of the error signals. We assume, however, that protected processes have reasonable signal handlers, which terminate the process after receiving such a signal. We include this outcome in the event *crash*.

The term *escape* describes a branch from the sequential flow of execution inside the random code sequence to any executable memory location. This event occurs when the instruction pointer (IP) is modified by random instructions to

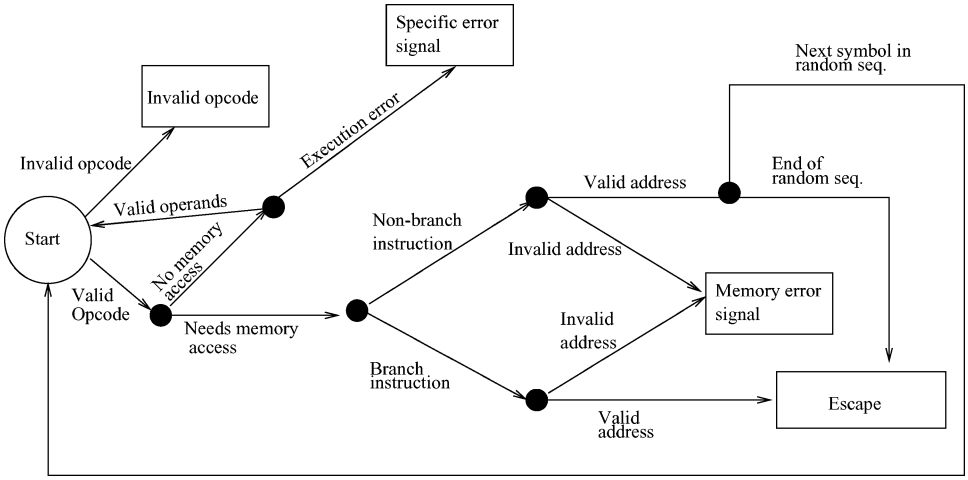

Fig. 1. State diagram for random code execution. The graph depicts the possible outcomes of executing a single random symbol. For variable-length instruction sets; the start state represents the reading of bytes until a nonambiguous decision about the identity of the symbol can be made.

point either to a location inside the executable code of the process, or to a location in a data section marked as executable even if it does not typically contain code.

An *error signal* is generated when the processor attempts to decode or execute a random symbol in the following cases:

(1) *Illegal instruction*: The symbol has no further ambiguity and it does not correspond to a defined instruction. The persymbol probability of this event depends solely on the density of the instruction set. An illegal instruction is signaled for undefined opcodes, illegal combinations of opcode and operand specifications, reserved opcodes, and opcodes undefined for a particular configuration (e.g., a 64-bit instruction on a 32-bit implementation of the PowerPC architecture).
(2) *Illegal read/write*: The instruction is legal, but it attempts to access a memory page to which it does not have the required operation privileges, or the page is outside the process' virtual memory.
(3) *Operation error*: Execution fails because the process state has not been properly prepared for the instruction; for example, division by 0, memory errors during a string operation, accessing an invalid port, or invoking a nonexistent interrupt.
(4) *Illegal branch*: The instruction is of the control transfer type and attempts to branch into a nonexecutable or nonallocated area.
(5) *Operation not permitted*: A legal instruction fails because the rights of the owner process do not allow its execution, for example, an attempt to use a privileged instruction in user mode.

There are several complications associated with branch instructions, depending on the target address of the branch. We assume that the only dangerous class of branch is a correctly invoked system call. The probability of randomly

invoking a system call in Linux is $\frac{1}{256} \times \frac{1}{256} \approx 1.52 \times 10^{-5}$ for IA32, and at most $\frac{1}{2^{32}} \approx 2.33 \times 10^{-10}$ for the 32-bit PowerPC. This is without adding the restriction that the arguments be reasonable. Alternatively, a process failure could remain hidden from an external observer, and we will see that this event is more likely.

A branch into the executable code of the process (ignoring alignment issues) will likely result in the execution of at least some instructions, and will perhaps lead to an infinite loop. This is an undesirable event because it hides the attack attempt even if it does not damage permanent data structures. We model successful branches into executable areas (random or nonrandom) as always leading to the *escape* state in Figure 1. This conservative assumption allows us to estimate how many attack instances will not be immediately detected. These "escapes" do not execute hostile code. They are simply attack instances that are likely not to be immediately observed by an external process monitor. The probability of a branch resulting in a crash or an escape depends at least in part on the size of the executing process, and this quantity is a parameter in our calculations.

Different types of branches have different probabilities of reaching valid code. For example, if a branch has the destination specified as a full address constant (*immediate*) in the instruction itself, it will be randomized, and the probability of landing in valid code will depend only on the density of valid code in the total address space, which tends to be low. A return takes the branching address from the current stack pointer, which has a high probability of pointing to a real-process return address.

We model these many possibilities by dividing memory accesses, for both branch and nonbranch instructions into two broad classes:

(1) *Process-state dominated*: When the randomized exploit begins executing, the only part of the process that has been altered is the memory that holds the attack code. Most of the process state (e.g., the contents of the registers, data memory, and stack) remains intact and consistent. However, we do not have good estimates of the probability that using these values from registers and memory will cause an error. So, we arbitrarily assign probabilities for these values and explore the sensitivity of the system to different probabilities. Experimentally we know that most memory accesses fail (see Figure 2).
(2) *Immediate dominated*: If a branch calculates the target address based on a full-address size immediate, we can assume that the probability of execution depends on the memory occupancy of the process, because the immediate is just another random number generated by the application of the mask to the attack code.

We use this classification in empirical studies of random code execution (Section 4.2). These experiments provide evidence that most processes terminate quickly when random code sequences are inserted. We then describe a theoretical model for the execution of random IA32 and PowerPC instructions (Section 5), which allows us to validate the experiments and provides a framework for future analysis of other architectures.

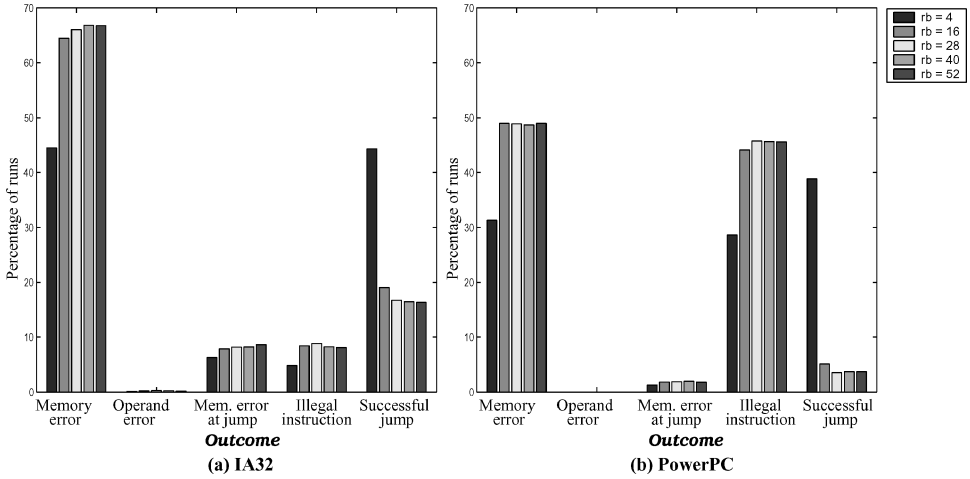

Fig. 2. Executing random blocks on native processors. The plots show the distribution of runs by type of outcome for (a) IA32 and (b) Power PC. Each color corresponds to a different random block size (*rb*): 4, 16, 28, 40, and 52 bytes. The filler is set such that the total process density is 5% of the possible 2^{32} address space. The experiment was run under the Linux operating system.

4.2 Empirical Testing

We performed two kinds of experiments: (1) execution of random blocks of bytes on native processors, and (2) execution of real attacks in RISE on IA32.

4.2.1 *Executing Blocks of Random Code.* We wrote a simple C program that executes blocks of random bytes. The block of random bytes simulates a randomized exploit running under RISE. We then tested the program for different block sizes (the "exploit") and different degrees of process space occupancy. The program allocates a prespecified amount of memory (determined by the *filler size* parameter) and fills it with the machine code for no operation (NOP). The block of random bytes is positioned in the middle of the filler memory.

Figure 2 depicts the observed frequency of the events defined in Section 4.1. There is a preponderance of memory access errors in both architectures, although the less dense PowerPC has an almost equal frequency of illegal instructions. Illegal instructions occur infrequently in the IA32 case. In both architectures, about one-third of legal branch instructions fail because of an invalid memory address, and two-thirds manage to execute the branch. Conditional branches form the majority of branch instructions in most architectures, and these branches have a high probability of executing because of their very short relative offsets.

Because execution probabilities could be affected by the memory occupancy of the process, we tested different process memory sizes. The process sizes used are expressed as fractions of the total possible 2^{32} address space (Table IV).

Each execution takes place inside GDB (the GNU debugger), single stepping until either a signal occurs or more than 100 instructions have been executed. We collect information about type of instruction, addresses, and types of signals during the run. We ran this scenario with 10,000 different seeds, five random

Table IV. Process Memory Densities (Relative to Process Size)

Process Memory Density (as a fraction of 2^{32} bytes)	0.0002956	0.0036093	0.0102365	0.0234910	0.0500000

Values are expressed as fractions of the total possible 2^{32} address space. They are based on observed process memory used in two busy IA32 Linux systems over a period of two days.

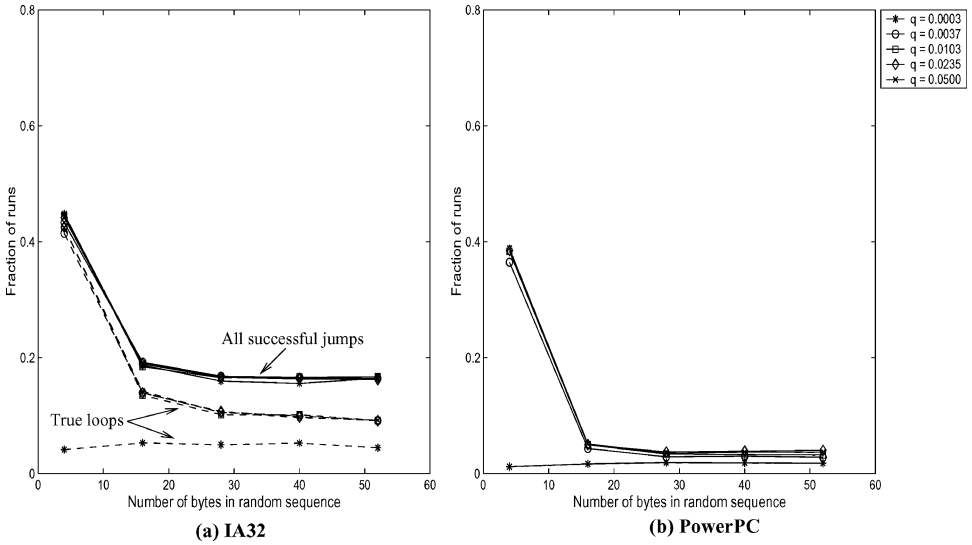

Fig. 3. Probability that random code escapes when executed for different block sizes (the x-axis) for (a) IA32 and (b) Power PC. Block size is the length of the sequence of random bytes inserted into the process. Each set of connected points represents a different memory density (q). Solid lines represent the fraction of runs that escaped under our definition of escape, and dotted lines show the fraction of "true" escaped executions (those that did not fail after escaping from the exploit area).

block sizes (4, 8, 24, 40, and 56 bytes), and five total process densities (see Table IV), both for the PowerPC and the IA32.

Figure 3 plots the fraction of runs that *escaped* according to our definition of escape (given in Section 4.1) for different memory densities. An execution was counted as an escape if a jump was executed and did not fail immediately (that is, it jumped to an executable section of the code). In addition, it shows the proportion of escapes that did not crash within a few bytes of the exploit area ("true" escapes: for example when the execution is trapped into an infinite loop). Escapes that continued executing for more than 100 instructions were terminated. The figure shows that for realistic block sizes (over 45 bytes), the proportion of true escapes is under 10% (IA32). In the Power PC case, although the fraction of escaped runs is smaller, most of the escapes do not fail afterwards, so the curves overlap.

A second observation (not shown) is that memory density has a negligible effect on the probability of escape, even though we created an environment that maximizes successful escapes. This is likely because the process sizes are still relatively small compared to the total address space and because only a minor-

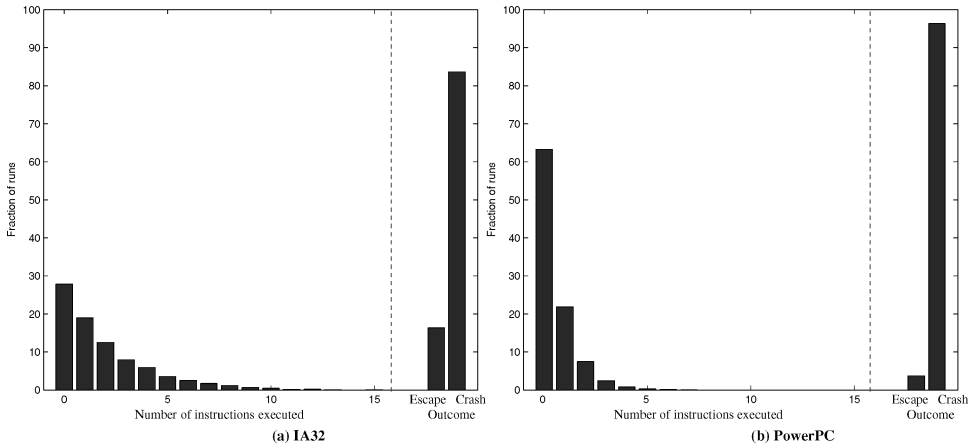

Fig. 4. Proportion of runs that fail after exactly n instructions, with memory density 0.05, and random block (simulated attack) size 52 bytes, for (a) IA32 and (b) PowerPC. On the right, the proportion of escaped vs. crashed runs is presented for comparison.

ity of memory accesses are affected by this density (those that are immediate dominated).

Figure 4 shows the proportion of failed runs that die after executing exactly n instructions. On the right side of the graph, the proportion of escaped versus failed runs is shown for comparison. Each instruction length bar comprises five subbars, one for each simulated attack size. We plot them all to show that the size of the attack has almost no effect on the number of instructions executed, except for very small sizes. On the IA32, more than 90% of all failed runs died after executing at most 6 instructions and in no case did the execution continue for more than 23 instructions. The effect is even more dramatic on the Power PC, where 90% of all failed runs executed for fewer than 3 instructions, and the longest failed run executed only 10 instructions.

4.2.2 *Executing Real Attacks under RISE.* We ran several vulnerable applications under RISE and attacked them repeatedly over the network, measuring how long it took them to fail. We also tested the two synthetic attacks from Fayolle and Glaume [2002]. In this case the attack and the exploit are in the same program, so we ran them in RISE for 10,000 times each, collecting output from RISE. Table V summarizes the results of these experiments. The real attacks fail within an average of two to three instructions (column 4). Column 3 shows how many attack instances we ran (each with a different random seed for masking) to compute the average. As column 5 shows, most attack instances crashed instead of escaping. The synthetic attacks averaged just under two instructions before process failure. No execution of any of the attacks was able to spawn a shell.

Within the RISE approach, one could avoid the problem of accidentally viable code by mapping to a larger instruction set. The size could be tuned to reflect the desired percentage of incorrect unscramblings that will likely lead immediately to an illegal instruction.

Table V. Survival Time in Executed Instructions for Attack Codes in Real-Applications Running under RISE

Attack Name	Application	No. of Attacks	Avg. no. of Insns.	Crashed Before Escape (%)
Named NXT resource record overflow	Bind 8.2.1-7	101	2.24	85.14
rpc.statd format string	nfs-utils 0.1.6-2	102	2.06	85.29
Samba trans2 exploit	smbd 2.2.1a	81	3.13	73.00
Synthetic heap exploit	N/A	10,131	1.98	93.93
Synthetic stack exploit	N/A	10,017	1.98	93.30

Column 4 gives the average number of instructions executed before failure (for instances that did not "escape").
Column 5 summarizes the percentage of runs crashing (instead of "escaping").

5. RISE SAFETY: THEORETICAL ANALYSIS

This section develops theoretical estimates of RISE safety and compares them with the experiments reported in the previous section. A theoretical analysis is important for several reasons. Diversified code techniques of various sorts and at various levels are likely to become more common. We need to understand exactly how much protection they confer. In addition, it will be helpful to predict the effect of code diversity on new architectures before they are built. For example, analysis allows us to predict how much increase in safety could be achieved by expanding the size of the instruction space by a fixed amount.

In the case of a variable-size instruction set, such as the IA32, we compute the aggregate probabilities using a Markov chain. In the case of a uniform-length instruction set, such as the PowerPC, we can compute the probabilities directly.

5.1 IA32 Instruction Set

For the IA32 instruction set, which is a CISC architecture, we use the published instruction set specification [Intel Corporation 2004] to build a Markov chain used to calculate the escape probability of a sequence of m random bytes (with byte-length $b = 8$ bits). Our analysis is based on the graph of event categories shown in Figure 1, but it is specialized to include the byte-to-byte symbol recognition transitions. A transition is defined as the reading of a byte by the processor, and the states describe either specific positions within instructions or exceptions. Appendix A provides the specifics of this particular Markov chain encoding.

Apart from the complexity of encoding the large and diverse IA32 instruction set, the major difficulty in the model is the decision of what to do when a symbol crosses the boundary of the exploit area. It is conceivable that the result of the

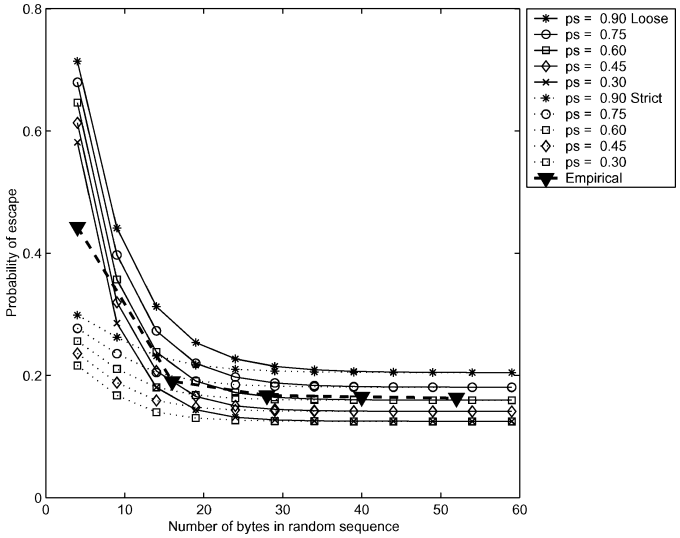

Fig. 5. Theoretical analysis of IA32 escape probability: The x-axis is the number of bytes in the random sequence, and the y-axis is the probability of escaping from a random string of m bytes. Each connected set of plotted points corresponds to one assumed probability of successfully executing a process-state-dominated memory access (p_s), with either strict or loose criterion of escape. The memory density is fixed at 0.05. For comparison with empirical data, the dashed line with triangles marks the observed average frequency of successful jumps (data taken from Figure 3 for the IA32 and memory density $q = 0.05$).

interpretation is still ambiguous at the byte that crosses the border. However, the model needs to decide what happens to the execution at that moment. We model this situation using both extremes: A *loose escape* declares the execution as an escape if the bytes interpreted up to the crossing point have not caused a crash; a *strict escape* declares that the execution of the partial instruction ends in crash. A characterization of the states in terms of the Markov chain is presented in Appendix A.

Figure 5 shows the probability of escape as a function of increasing sequence length for both loose and strict criteria of escape for a fixed memory density (0.05), and for different probabilities of a process state-dominated memory access to execute. The plot reveals several interesting phenomena.

First, the plots converge to a steady state quickly—in less than 20 bytes. This is consistent with the empirical data. Notably, the probability of escape converges to a nonzero value. This means that independently of exploit or process size, there will always be a nonzero probability that a sequence of random code will escape.

A second observation revealed by the plot is the relatively small difference between our loose and strict criteria for escape. The main difference between both cases is how to interpret the last instruction in the sequence if the string has not crashed before the exploit border. Not surprisingly, as sequences get longer, the probability of reaching the last symbol diminishes, so the overall effect of an ambiguous last instruction in those few cases is respectively smaller.

Table VI. Partition of Symbols into Disjoint Sets

Set Name	Type of Instructions in Set
U	Undefined instructions.
P	Privileged instructions.
B_{SR}	Small offset, relative branch
L_D	Legal instructions with no memory access and no branching. All branches require memory access, so L_D only contains linear instructions.
L_{MI}	Legal no-branch instructions with immediate-dominated memory access.
B_{MI}	Legal branch instructions with immediate-dominated memory-access.
L_{MP}	Legal no-branch instructions with process-state dominated memory-access.
B_{MP}	Legal branch instructions with process-state dominated memory access.

A third observation (data not shown in the figure) is that for different memory densities, the escape curves are nearly identical. This means that memory size has almost no effect on the probability of escape at typical process memory occupancies. In part, this reflects the fact that most jumps use process-state-dominated memory accesses. In particular, immediate-dominated memory accesses constitute a very small proportion of the instructions that use memory (only 4 out of more than 20 types of jumps).

The fourth observation concerns the fact that the first data point in the empirical run (block size of 4 bytes) differs markedly from all the strict and loose predicted curves. Both criteria are extreme cases and the observed behavior is in fact bounded by them. The divergence is most noticeable during the first 10 bytes, as most IA32 instructions have a length between 4 and 10 bytes. As noted before, the curves for loose and strict converge rapidly as the effect of the last instruction becomes less important, and so we see a much closer fit with the predicted behavior after 10 bytes, as the bounds become tighter.

The final observation is that the parameter p_s varies less than expected. We were expecting that the empirical data would have an ever-increasing negative slope, given that in principle the entropy of the process would increase as more instructions were executed. Instead, we get a close fit with $p_s = 0.6$ after the first 20 bytes. This supports our approximation to the probability of execution for process-state dominated instructions, as a constant that can be determined with system profiling.

5.2 Uniform-Length Instruction Set Model

The uniform-length instruction set is simpler to analyze because it does not require conditional probabilities on instruction length. Therefore, we can estimate the probabilities directly without resorting to a Markov chain. Our analysis generalizes to any RISC instruction set, but we use the PowerPC [IBM 2003] as an example.

Let all instructions be of length b bits (usually $b = 32$). We calculate the probability of escape from a random string of m symbols $r = r_1 \ldots r_m$, each of length b bits (assumed to be drawn from a uniform distribution of 2^b possible symbols). We can partition all possible symbols in disjoint sets with different execution characteristics. Table VI lists the partition we chose to use. Figure 7 in Appendix B illustrates the partition in terms of the classification of events

given in Section 4.1. $S = U \cup P \cup B_{SR} \cup L_D \cup L_{MI} \cup B_{MI} \cup L_{MP} \cup B_{MP}$ is the set of all possible symbols that can be formed with b bits. $|S| = 2^b$. The probability that a symbol s belongs to any given set I (where I can be any one of U, P, B_{SR}, L_D, L_{MI}, B_{MI}, L_{MP} or B_{MP}) is given by $P\{s \in I\} = P(I) = \frac{|I|}{2^b}$.

If there are a bits for addressing (and consequently the size of the address space is 2^a); E_I is the event that a symbol belonging to set I executes; M_t is the total memory space allocated to the process; M_e is the total executable memory of the process; and p_s is the probability that a memory access dominated by the processor state succeeds, then the probabilities of successful execution for instructions in each set are

For illegal and privileged opcodes, $P(E_U) = P(E_P) = 0$.

For the remaining legal opcodes, $P(E_{L_D}) = P(E_{B_{SR}}) = 1$; $P(E_{L_{MI}}) = \frac{M_t}{2^a}$; $P(E_{B_{MI}}) = \frac{M_e}{2^a}$; $P(E_{L_{MP}}) = p_s$ and $P(E_{B_{MP}}) = p_s$.

We are interested in the probability of a successful branch (escape) out of a sequence of n random bytes. Let X_n denote the event that an execution escapes at *exactly* symbol n. This event requires that $n-1$ instructions execute without branching and that the nth instruction branches successfully. In consequence, $P(X_n) = (P(L))^{n-1}P(E)$, where $P(L) = P(L_D) + P(L_{MI})P(E_{L_{MI}}) + P(L_{MP})$ is the probability that a symbol executes a successful linear instruction, and $P(E) = P(B_{MI})P(E_{B_{MI}}) + P(B_{MP}) + P(B_{SR})$ is the probability that a symbol executes a valid branch.

If $X_n^\star$ is the event that the execution of a random string $r = r_1 \cdots r_n$ escapes, its probability $P(X_n^\star)$ is given by (Appendix B:

$$P(X_n^\star) = P(E)\frac{1 - P(L)^{n+1}}{1 - P(L)} + P(L)^n$$

$P(X_n^\star)$ is plotted in Figure 6 for different values of p_s, increasing random code sizes and a given memory density (0.05 as in the IA32 case). The comparable data points from our experiments are shown for comparison. We did not plot results for different memory densities because the difference among the curves is negligible. The figure shows that the theoretical analysis agrees with our experimental results. The parameters were calculated from the published documentation of the PowerPC instruction set [IBM 2003], for the 32-bit case: $b = 32$, $a = 32$, $P(L_D) \approx 0.25$, $P(L_{MI}) = 0$, $P(L_{MP}) \approx 0.375$, $P(B_{MI}) \approx 0.015$, $P(B_{MP}) \approx 0.030$, $P(B_{SR}) \approx 0.008$.

It can be seen that the probability of escape converges to a nonzero value. For a uniform-length instruction set, this value can be calculated as

$$\lim_{n\to\infty} P(X_n^\star) = \frac{P(E)}{1 - P(L)}.$$

The limit value of $P(X_n^\star)$ is the lower bound on the probability of a sequence of length n escaping. It is independent of n, so larger exploit sizes are no more likely to fail than smaller ones in the long run. It is larger than 0 for any architecture in which the probability of successful execution of a jump to a random location is larger than 0.

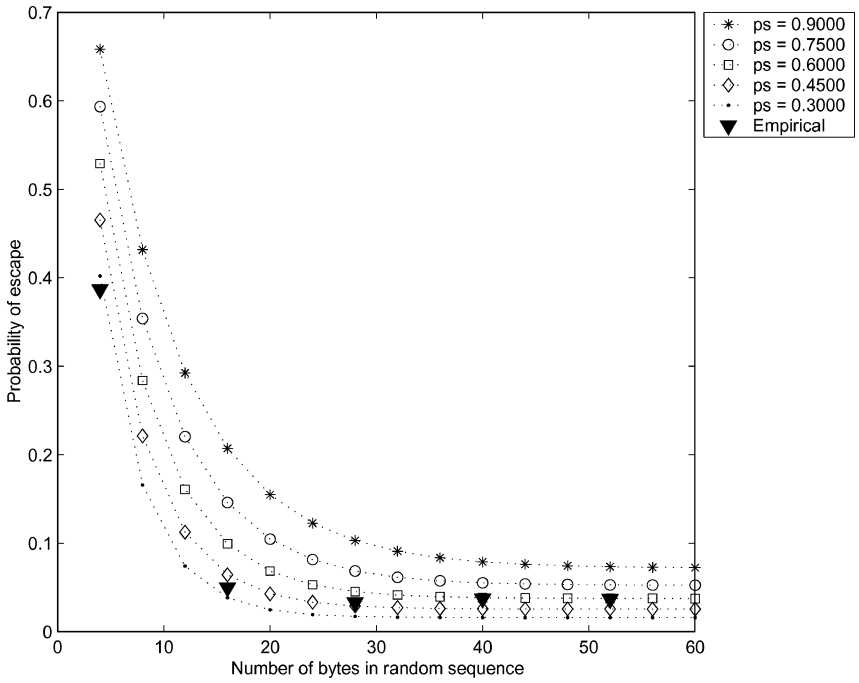

Fig. 6. Theoretical probability of escape for a random string of n symbols. Each curve plots a different probability of executing a process-state-determined memory access (p_s) for the PowerPC uniform-length instruction set. Process memory occupancy is fixed at 0.05. The large triangles are the measured data points for the given memory occupancy (data taken from Figure 3 for the PowerPC and memory density $q = 0.05$), and the dotted lines are the predicted probabilities of escape.

6. RELATED WORK

Our randomization technique is an example of *automated diversity*, an idea that has long been used in software engineering to improve fault tolerance [Avizienis 1995; Avizienis and Chen 1977; Randell 1975] and more recently has been proposed as a method for improving security [Cohen 1993; Forrest et al. 1997; Pu et al. 1996]. The RISE approach was introduced in Barrantes et al. [2003], and an approach similar to RISE was proposed in Kc et al. [2003].

Many other approaches have been developed for protecting programs against particular methods of code injection, including: static code analysis [Dor et al. 2003; Larochelle and Evans 2001; Wagner et al. 2000] and runtime checks, using either static code transformations [Avijit et al. 2004; Baratloo et al. 2000; Chiueh and Hsu 2001; Cowan et al. 1998, 2001; Etoh and Yoda 2000, 2001; Jones and Kelly 1997; Lhee and Chapin 2002; Nebenzahl and Wool 2004; Prasad and Chiueh 2003; Ruwase and Lam 2004; Tsai and Singh 2001; Vendicator 2000; Xu et al. 2002], dynamic instrumentation [Baratloo et al. 2000; Kiriansky et al. 2002], or hybrid schemes [Jim et al. 2002; Necula et al. 2002]. In addition, some methods focus on protecting an entire system rather than a particular program, resulting in defense mechanisms at the operating system level and

hardware support [Milenković et al. 2004; PaX Team 2003; Xu et al. 2002]. Instruction-set randomization is also related to hardware code encryption methods explored in Kuhn [1997] and those proposed for TCPA/TCG [TCPA 2004].

6.1 Automated Diversity

Diversity in software engineering is quite different from diversity for security. In software engineering, the basic idea is to generate multiple independent solutions to a problem (e.g., multiple versions of a software program) with the hope that they will fail independently, thus greatly improving the chances that some solution out of the collection will perform correctly in every circumstance. The different solutions may or may not be produced manually, and the number of solutions is typically quite small, around 10.

Diversity in security is introduced for a different reason. Here, the goal is to reduce the risk of widely replicated attacks, by forcing the attacker to redesign the attack each time it is applied. For example, in the case of a buffer overflow attack, the goal is to force the attacker to rewrite the attack code for each new computer that is attacked. Typically, the number of different diverse solutions is very high, potentially equal to the total number of program copies for any given program. Manual methods are thus infeasible, and the diversity must be produced automatically.

Cowan et al. [2000] introduced a classification of diversity methods applied to security (called "security adaptations") which classifies diversifications based on what is being adapted—either the interface or the implementation. Interface diversity modifies code layout or access controls to interfaces, without changing the underlying implementation to which the interface gives access. Implementation diversity, on the other hand, modifies the underlying implementation of some portion of the system to make it resistant to attacks. RISE can be viewed as a form of interface diversity at the machine code level.

In 1997, Forrest et al. presented a general view of the possibilities of diversity for security [Forrest et al. 1997], introducing the idea of deliberately diversifying data and code layouts. They used the example of randomly padding stack frames to make exact return address locations less predictable, and thus more difficult for an attacker to locate. Developers of buffer overflow attacks have developed a variety of workarounds—such as "ramps" and "landing zones" of no-ops and multiple return addresses. Automated diversity via random stack padding coerces an attacker to use such techniques; it also requires larger attack codes in proportion to the size range of random padding employed.

Other work in automated diversity for security has also experimented with diversifying data layouts [Cohen 1993; Pu et al. 1996], as well as system calls [Chew and Song 2002], and file systems [Cowan et al. 2000]. In addition, several projects address the code-injection threat model directly, and we describe those projects briefly.

Chew and Song [2002] proposed a method that combines kernel and loader modification on the system level with binary rewriting at the process level

to provide system call number randomization, random stack relocation, and randomization of standard library calls. This work has not been completely evaluated to our knowledge.

Address space layout randomization (ASLR) [PaX Team 2003] and transparent runtime randomization (TRR) [Xu et al. 2003] randomize the positions of the stack, shared libraries, and heap. The main difference between the two is the implementation level. ASLR is implemented in the kernel, while TRR modifies the loader program. Consequently, TRR is more oriented to the end user.

Bhatkar et al. [2003] describe a method that randomizes the addresses of data structures internal to the process, in addition to the base address of the main segments. Internal data and code blocks are permuted inside the segments and the guessing range is increased by introducing random gaps between objects. The current implementation instruments object files and ELF binaries to carry out the required randomizations. No access to the source code is necessary, but this makes the transformations extremely conservative. This technique nicely complements that of RISE, and the two could be used together to provide protection against both code injection and return-into-libc attacks simultaneously.

PointGuard [Cowan et al. 2003] uses automated randomization of pointers in the code and is implemented by instrumenting the intermediate code (AST in GCC).

The automated diversity project that is closest to RISE is the system described in Kc et al. [2003], which also randomizes machine code. There are several interesting points of comparison with RISE, and we describe two of them: (1) persystem (whole image) versus perprocess randomization; (2) Bochs [Butler 2004] versus Valgrind as emulator. First, in the Kc et al. implementation, a single key is used to randomize the image, all the libraries, and any applications that need to be accessed in the image. The system later boots from this image. This has the advantage that in theory, kernel code could be randomized using their method although most code-injection attacks target application code. A drawback of this approach lies in its key management. There is a single key for all applications in the image, and the key cannot be changed during the lifetime of the image. Key guessing is a real possibility in this situation, because the attacker would be likely to know the cleartext of the image. However, the Kc et al. system is more compact because there is only one copy of the libraries. On the other hand, if the key is guessed for any one application or library, then all the rest are vulnerable. Second, the implementations differ in their choice of emulator. Because Bochs is a pure interpreter it incurs a significant performance penalty, while emulators such as Valgrind can potentially achieve close-to-native efficiency through the use of optimized and cached code fragments.

A randomization of the SQL language was proposed in Boyd and Keromytis [2004]. This technique is essentially the same one used in the Perl randomizer [Kc et al. 2003], with a random string added to query keywords. It is implemented through a proxy application on the server side. In principle, there could be one server proxy per database connection, thus allowing more key

diversity. The performance impact is minimal, although key capture is theoretically possible in a networked environment.

6.2 Other Defenses Against Code Injection

Other defenses against code injection (sometimes called "restriction methods") can be divided into methods at the program and at the system level. In turn, approaches at the program level comprise static code analysis and runtime code instrumentation or surveillance. System level solutions can be implemented in the operating system or directly through hardware modifications. Of these, we focus on the methods most relevant to RISE.

6.2.1 *Program-Level Defenses Against Code Injection.* Program-level approaches can be seen as defense-in-depth, beginning with suggestions for good coding practices and/or use of type-safe languages, continuing with automated analysis of source code, and finally reaching static or dynamic modification of code to monitor the process progress and detect security violations. Comparative studies on program-level defenses against buffer overflows have been presented by Fayolle and Glaume [2002], Wilander and Kamkar [2003], and Simon [2001]. Several relevant defenses are briefly discussed below.

The StackGuard system [Cowan et al. 1998] modifies GCC to interpose a a canary word before the return address, the value of which is checked before the function returns. An attempt to overwrite the return address via linear stack smashing will change the canary value and thus be detected.

StackShield [Vendicator 2000], RAD [Chiueh and Hsu 2001], install-time vaccination [Nebenzahl and Wool 2004], and binary rewriting[Prasad and Chiueh 2003] all use instrumentations to store a copy of the function return address off the stack and check against it before returning to detect an overwrite. Another variant, Propolice [Etoh and Yoda 2000, 2001] uses a combination of a canary word and frame data relocation to avoid sensible data overwriting. Split control and data stack [Xu et al. 2002] divides the stack in a control stack for return addresses and a data stack for all other stack-allocated variables.

FormatGuard [Cowan et al. 2001] used the C preprocessor (CPP) to add parameter-counting to printf-like C functions and defend programs against format print vulnerabilities. This implementation was not comprehensive even against this particular type of attacks.

A slightly different approach uses wrappers around standard library functions, which have proven to be a continuous source of vulnerabilities. Libsafe [Baratloo et al. 2000; Tsai and Singh 2001], TIED, and LibsafePlus [Avijit et al. 2004], and the type-assisted bounds checker proposed by Lhee and Chapin [2002] intercept library calls and attempt to ensure that their manipulation of user memory is safe.

An additional group of techniques depends on runtime bounds checking of memory objects, such as the Kelly and Jones bound checker [Jones and Kelly 1997] and the recent C range error detector (CRED) [Ruwase and Lam 2004]. Their heuristics differ in the way of determining if a reference is still legal. Both can generate false positives, although CRED is less computationally expensive.

The common theme in all these techniques is that they are specific defenses, targeting specific points of entry for the injected code (stack, buffers, format functions, and so on). Therefore, they cannot prevent an injection arriving from a different source or an undiscovered vulnerability type. RISE, on the other hand, is a generic defense that is independent of the method by which binary code is injected.

There is also a collection of dynamic defense methods which do not require access to the original sources or binaries. They operate directly on the process in memory, either by inserting instrumentation as extra code (during the load process or as a library) or by taking complete control as in the case of native-to-native emulators.

Libverify [Baratloo et al. 2000] saves a copy of the return address to compare at the function end, so it is a predecessor to install-time vaccination [Nebenzahl and Wool 2004] and binary rewriting [Prasad and Chiueh 2003], with the difference that it is implemented as a library that performs the rewrite dynamically, so the binaries on disk do not require modification.

Code shepherding [Kiriansky et al. 2002] is a comprehensive, policy-based restriction defense implemented over a binary-to-binary optimizing emulator. The policies concern client code control transfers that are intrinsically detected during the interpretation process. Two of those types of policies are relevant to the RISE approach.

Code origin policies grant differential access based on the source of the code. When it is possible to establish if the instruction to be executed came from a disk binary (modified or unmodified) or from dynamically generated code (original or modified after generation), policy decisions can be made based on that origin information. In our model, we are implicitly implementing a code origin policy, in that only unmodified code from disk is allowed to execute. An advantage of the RISE approach is that the origin check cannot be avoided—only properly sourced code is mapped into the private instruction set so it executes successfully. Currently, the only exception we have to the disk origin policy is for the code deposited in the stack by signals. RISE inherits its signal manipulation from Valgrind [Nethercote and Seward 2003]. More specifically, all client signals are intercepted and treated as special cases. Code left on the stack is executed separately from the regular client code fetch cycle so it is not affected by the scrambling. This naturally resembles PaX's special handling of signals, where code left on the stack is separately emulated.

Also relevant are restricted control transfers in which a transfer is allowed or disallowed according to its source, destination, and type. Although we use a restricted version of this policy to allow signal code on the stack, in most other cases we rely on the RISE language barrier to ensure that injected code will fail.

6.2.2 *System-Level Defenses Against Code Injection.* System level restriction techniques can be applied in the operating system, hardware, or both. We briefly review some of the most important system-level defenses.

The nonexecutable stack and heap as implemented in the PAGEEXEC feature of PaX [PaX Team 2003] is hardware assisted. It divides allocation into data and code TLBs and intercepts all page-fault handlers into the code TLB.

As with any hardware-assisted technique, it requires changes to the kernel. RISE is functionally similar to these techniques, sharing the ability to randomize ordinary executable files with no special compilation requirements. Our approach differs, however, from nonexecutable stacks and heaps in important ways. First, it does not rely on special hardware support (although RISE pays a performance penalty for its hardware independence). Second, although a system administrator can choose whether to disable certain PaX features on a perprocess basis, RISE can be used by an end-user to protect user-level processes without any modification to the overall system.

A third difference between PaX and RISE is in how they handle applications that emit code dynamically. In PaX, the process-emitting code requires having the PAGEEXEC feature disabled (at least), so the process remains vulnerable to injected code. If such a process intended to use RISE, it could modify the code-emitting procedures to use an interface provided by RISE, and derived from Valgrind's interface for Valgrind-aware applications. The interface uses a validation scheme based on the original randomization of code from disk. In a pure language randomization, a process-emitting dynamic code would have to do so in the particular language being used at that moment. In our approximation, the process using the interface scrambles the new code before execution. The interface, a RISE function, considers the fragment of code as a new library, and randomizes it accordingly. In contrast to nonexecutable stack/heap, this does not make the area where the new code is stored any more vulnerable, as code injected in this area will still be expressed in nonrandomized code and will not be able to execute except as random bytes.

Some other points of comparison between RISE and PaX include

(1) *Resistance to return-into-libc*: Both RISE and PaX PAGEEXEC features are susceptible to return-into-libc attacks when implemented as an isolated feature. RISE is vulnerable to return-into-libc attacks without an internal data structure randomization, and data structure randomization is vulnerable to injected code without the code randomization. Similarly, as the PaX Team notes, LIBEEXEC is vulnerable to return-into-libc without ASLR (automatic stack and library randomization), and ASLR is vulnerable to injected code without PAGEEXEC [PaX Team 2003]. In both cases, the introduction of the data structure randomization (at each corresponding granularity level) makes return-into-libc attacks extremely unlikely.
(2) *Signal code on the stack*: Both PaX and RISE support signal code on the stack. They both treat it as a special case. RISE in particular is able to detect signal code as it intercepts all signals directed to the emulated process and examines the stack before passing control to the process.
(3) *C trampolines*: PaX detects trampolines by their specific code pattern and executes them by emulation. The current RISE implementation does not support this, although it would not be difficult to add it.

StackGhost [Frantzen and Shuey 2001] is a hardware-assisted defense implemented in OpenBSD for the Sparc architecture. The return address of

functions is stored in registers instead of the stack, and for a large number of nested calls StackGhost protects the overflowed return addresses through write protection or encryption.

Milenković et al. [2004] propose an alternative architecture where linear blocks of instructions are signed on the last basic block (equivalent to a line of cache). The signatures are calculated at compilation time and loaded with the process into a protected architectural structure. Static libraries are compiled into a single executable with a program, and dynamic libraries have their own signature file loaded when the library is loaded. Programs are stored unmodified, but their signature files should be stored with strong cryptographic protection. Given that the signatures are calculated once, at compile time, if the signature files are broken, the program is vulnerable.

Xu et al. [2002] propose using a secure return address stack (SRAS) that uses the redundant copy of the return address maintained by the processor's fetch mechanism to validate the return address on the stack.

6.3 Hardware Encryption

Because RISE uses runtime code scrambling to improve security, it resembles some hardware-based code encryption schemes. Hardware components to allow decryption of code and/or data on-the-fly have been proposed since the late 1970s [Best 1979, 1980] and implemented as microcontrollers for custom systems (for example, the DS5002FP microcontroller [Dallas Semiconductor 1999]). The two main objectives of these cryptoprocessors are to protect code from piracy and data from in-chip eavesdropping. An early proposal for the use of hardware encryption in general-purpose systems was presented by Kuhn [1997] for a very high threat level where encryption and decryption were performed at the level of cache lines. This proposal adhered to the model of protecting licensed software from users, and not users from intruders, so there was no analysis of shared libraries or how to encrypt (if desired) existing open applications. A more extensive proposal was included as part of TCPA/TCG [TCPA 2004]. Although the published TCPA/TCG specifications provide for encrypted code in memory, which is decrypted on the fly, TCPA/TCG is designed as a much larger authentication and verification scheme and has raised controversies about digital rights management (DRM) and end-users' losing control of their systems [Anderson 2003; Arbaugh 2002]. RISE contains none of the machinery found in TCPA/TCG for supporting DRM. On the contrary, RISE is designed to maintain control locally to protect the user from injected code.

7. DISCUSSION

The preceding sections describe a prototype implementation of the RISE approach and evaluate its effectiveness at disrupting attacks. In this section, we address some larger questions about RISE.

7.1 Performance Issues

Although Valgrind has some limitations, discussed in Section 2, we are optimistic that improved designs and implementations of "randomized machines"

would improve performance and reduce resource requirements, potentially expanding the range of attacks the approach can mitigate. We have also observed that even in its current version, the performance RISE offers could be acceptable if the processes are I/O bound and/or use the network extensively.

In the current implementation, RISE safety is somewhat limited by the dense packing of legal IA32 instructions in the space of all possible byte patterns. A random scrambling of bits is likely to produce a different legal instruction. Doubling the size of the instruction encoding would enormously reduce the risk of a processor's successfully executing a long enough sequence of unscrambled instructions to do damage. Although our preliminary analysis shows that this risk is low even with the current implementation, we believe that emerging soft-hardware architectures such as Crusoe [Klaiber 2000] will make it possible to reduce the risk even further.

7.2 Is RISE Secure?

A valid concern when evaluating RISE's security is its susceptibility to key discovery, as an attacker with the appropriate scrambling information could inject scrambled code that will be accepted by the emulator. We believe that RISE is highly resistant to this class of attack.

RISE is resilient against brute force attacks because the attacker's work is exponential in the shortest code sequence that will make an externally detectable difference if it is unscrambled properly. We can be optimistic because most IA32 attack codes are at least dozens of bytes long, but if a software flaw existed that was exploitable with, say, a single 1-byte opcode, then RISE would be vulnerable, although the process of guessing even a 1-byte representation would cause system crashes easily detectable by an administrator.

An alternative path for an attacker is to try to inject arbitrary address ranges of the process into the network, and recover the key from the downloaded information. The download could be part of the key itself (stored in the process address space), scrambled code, or unscrambled data. Unscrambled data does not give the attacker any information about the key. Even if the attacker could obtain scrambled code or pieces of the key (they are equivalent because we can assume that the attacker has knowledge of the program binary), using the stolen key piece might not be feasible. If the key is created eagerly, with a key for every possible address in the program, past or future, then the attacker would still need to know where the attack code is going to be written in process space to be able to use that information. However, in our implementation, where keys are created lazily for code loaded from disk, the key for the addresses targeted by the attack might not exist, and therefore might not be discoverable. The keys that do exist are for addresses that are usually not used in code injection attacks because they are write protected. In summary, it would be extremely difficult to discover or use a particular encoding during the lifetime of a process.

Another potential vulnerability is RISE itself. We believe that RISE would be difficult to attack for several reasons. First, we are using a network-based threat model (attack code arrives over a network) and RISE does not perform

network reads. In fact it does not read any input at all after processing the run arguments. Injecting an attack through a flawed RISE read is thus impossible.

Second, if an attack arises inside a vulnerable application and the attacker is aware that the application is being run under RISE, the vulnerable points are the code cache and RISE's stack, as an attacker could deposit code and wait until RISE proceeds to execute something from these locations. Although RISE's code is not randomized because it has to run natively, the entire area is write protected, so it is not a candidate for injection. The cache is read-only during the time that code blocks are executed, which is precisely when this hypothetical attack would be launched, so injecting into the cache is infeasible.

Another possibility is a *jump-into-RISE* attack. We consider three ways in which this might happen:[3]

(1) The injected address of RISE code is in the client execution path cache.
(2) The injected address of RISE code is in the execution path of RISE itself.
(3) The injected address of RISE code is in a code fragment in the cache.

In case 1, the code from RISE will be interpreted. However, RISE only allows certain self-functions to be called from client code, so everything else will fail. Even for those limited cases, RISE checks the call origin, disallowing any attempt to modify its own structures.

For case 2, the attacker would need to inject the address into a RISE data area in RISE's stack or in an executable area. The executable area is covered by case 3. For RISE's data and stack areas we have introduced additional randomizations. The most immediate threat is the stack, so we randomize its start address. For other data structures, the location could be randomized using the techniques proposed in Bhatkar et al. [2003], although this is unimplemented in the current prototype. Such a randomization would make it difficult for the attacker to guess its location correctly. An alternative, although much more expensive, solution would be to monitor all writes and disallow modifications from client code and certain emulator areas.

It is worth noting that this form of attack (targeting emulator data structures) would require executing several commands without executing a single machine language instruction. Although such attacks are theoretically possible via chained system calls with correct arguments, and simple (local) attacks have been shown to work [Nergal 2001], these are not a common technique [Wilander and Kamkar 2003]. In the next version of RISE, we plan to include full data structure address randomization, which would make these rare attacks extremely difficult to execute.

Case 3 is not easily achieved because fragments are write protected. However, an attacker could conceivably execute an *mprotect* call to change writing rights and then write the correct address. In such a case, the attack would execute. This is a threat for applications running over emulators, as it undermines all other security policies [Kiriansky et al. 2002]. In the current RISE implementation, we borrow the solution used in Kiriansky et al. [2002], monitoring

[3]We rely on the fact that RISE itself does not receive any external input once it is running.

all calls to the *mprotect* system call by checking their source and destination and not allowing executions that violate the protection policy.

7.3 Code/Data Boundaries

An essential requirement for using RISE for improving security is that the distinction between code and data must be carefully maintained. The discovery that code and data can be systematically interchanged was a key advance in early computer design, and this dual interpretation of bits as both numbers and commands is inherent to programmable computing. However, all that flexibility and power turn into security risks if we cannot control how and when data become interpreted as code. Code-injection attacks provide a compelling example, as the easiest way to inject code into a binary is by disguising it as data, for example, as inputs to functions in a victim program.

Fortunately, code and data are typically used in very different ways, so advances in computer architecture intended solely to improve performance, such as separate instruction caches and data caches, also have helped to enforce good hygiene in distinguishing machine code from data, helping make the RISE approach feasible. At the same time, of course, the rise of mobile code, such as Javascript in web pages and macros embedded in word processing documents, tends to blur the code/data distinction and create new risks.

7.4 Generality

Although our paper illustrates the idea of randomizing instruction sets at the machine-code level, the basic concept could be applied wherever it is possible to (1) distinguish code from data, (2) identify all sources of trusted code, and (3) introduce hidden diversity into all and only the trusted code. A RISE for protecting `printf` format strings, for example, might rely on compile-time detection of legitimate format strings, which might either be randomized upon detection, or flagged by the compiler for randomization sometime closer to runtime. Certainly, it is essential that a running program interact with external information, at some point, or no externally useful computation can be performed. However, the recent SQL attacks illustrate the increasing danger of expressing running programs in externally known languages [Harper 2002]. Randomized instruction set emulators are one step toward reducing that risk.

An attraction of RISE, compared to an approach such as code shepherding, is that injected code is stopped by an inherent property of the system, without requiring any explicit or manually defined checks before execution. Although divorcing policy from mechanism (as in code shepherding) is a valid design principle in general, complex user-specified policies are more error prone than simple mechanisms that hard code a well-understood policy.

8. CONCLUSIONS

In this paper we introduced the concept of a randomized instruction set emulator as a defense against binary code injection attacks. We demonstrated the feasibility and utility of this concept with a proof-of-concept implementation based

on Valgrind. Our implementation successfully scrambles binary code at load time, unscrambles it instruction-by-instruction during instruction fetch, and executes the unscrambled code correctly. The implementation was successfully tested on several code-injection attacks, some real and some synthesized, which exhibit common injection techniques.

We also addressed the question of RISE safety—how likely are random byte sequences to cause damage if executed. We addressed this question both experimentally and theoretically and conclude that there is an extremely low probability that executing a sequence of random bytes would cause real damage (say by executing a system call). However, there is a slight probability that such a random sequence might escape into an infinite loop or valid code. This risk is much lower for the Power PC instruction set than it is for the IA32, due to the density of the IA32 instruction set. We thus conclude that a RISE approach would be even more successful on the Power PC architecture than it is on the IA32.

As the complexity of systems grows, and 100% provable overall system security seems an ever more distant goal, the principle of diversity suggests that having a variety of defensive techniques based on different mechanisms with different properties stands to provide increased robustness, even if the techniques address partially or completely overlapping threats. Exploiting the idea that it is hard to get much done when you do not know the language, RISE is another technique in the defender's arsenal against binary code injection attacks.

APPENDIX

A. ENCODING OF THE IA32 MARKOV CHAIN MODEL

In this appendix, we discuss the details for the construction of the Markov chain representing the state of the processor as each byte is interpreted.

If $X_t = j$ is the event of being in state j at time t (in our case, at the reading of byte t), the transition probability $P\{X_{t+1} = j | X_t = i\}$ is denoted p_{ij} and is the probability that the system will be in state j at byte $t + 1$ if it is in state i for byte t.

For example, when the random sequence starts (in state $start$), there is some probability p that the first byte will correspond to an existing 1-byte opcode that requires an additional byte to specify memory addressing (the Mod-Reg-R/M (MRM) byte). Consequently, we create a transition from *start* to *mrm* with some probability p: $p_{start,mrm} = p$. p is the number of instructions with one opcode that require the MRM byte, divided by the total number of possibilities for the first byte (256). In IA32 there are 41 such instructions, so $p_{start,mrm} = \frac{41}{256}$.

If the byte corresponds to the first byte of a 2-byte instruction, we transition to an intermediate state that represents the second byte of that family of instructions, and so on. There are two exit states: *crash* and *escape*. The *crash* state is reached when an illegal byte is read, or there is an attempt to use invalid memory, for an operation or a jump. The second exit state, *escape*, is reached probabilistically when a legitimate jump is executed. This is related to the escape event.

Because of the complexity of the IA32 instruction set, we simplified in some places. As far as possible, we adhered to the worst-case principle, in which we overestimated the bad outcomes when uncertainty existed (e.g., finding a legal instruction, executing a privileged instruction, or jumping). The next few paragraphs describe these simplifications.

We made two simplifications related to instructions. The IA32 has instruction modifiers called prefixes that can generate complicated behaviors when used with the rest of the instruction set. We simplified by treating all of them as independent instructions of length 1 byte, with no effect on the following instructions. This choice overestimates the probability of executing those instructions, as some combinations of prefixes are not allowed, others significantly restrict the kind of instructions that can follow, or make the addresses or operands smaller. In the case of regular instructions that require longer low-probability pathways, we combined them into similar patterns. Privileged instructions are assumed to fail with probability of 1.0 because we assume that the RISE-protected process is running at user level.

In the case of conditional branches, we assess the probability that the branch will be taken, using the combination of flag bits required for the particular instruction. For example, if the branch requires that two flags have a given value (0 or 1), the probability of taking the branch is set to 0.25 . A nontaken branch transitions to the *start* state as a linear instruction. All conditional branches in IA32 use *relative* (to the current Instruction Pointer), 8- or 16-bit displacements. Given that the attack had to be in an executable area to start with, this means that it is likely that the jump will execute. Consequently, for conditional branches we transition to *escape* with probability 1. This is consistent with the observed behavior of successful jumps.

A.1 Definition of Loose and Strict Criteria of Escape

Given that the definition of escape is relative to the position of the instruction in the exploit area, it is necessary to arbitrarily decide if to classify an incomplete interpretation as an escape or as a crash. This is the origin of the loose and strict criteria.

In terms of the Markov chain, the loose and strict classifications are defined as follows:

(1) *Loose escape*: Starting from the *start* state, reach any state except *crash*, in m transitions (reading m bytes).
(2) *Strict escape*: Reach the *escape* state in m or fewer transitions from the *start* state (in m bytes).

If T is the transition matrix representing the IA32 Markov chain, then to find the probability of escape from a sequence of m random bytes, we need to determine if the chain is in state *start* or *escape* (the strict criterion) or not in state *crash* (the loose criterion) after advancing m bytes. These probabilities are

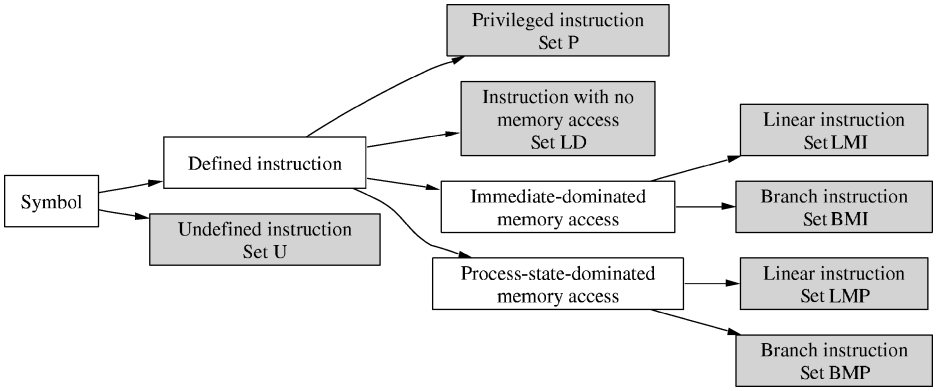

Fig. 7. Partition of symbols into disjoint sets based on the possible outcome paths of interest in the decoding and execution of a symbol. Each path defines a set. Each shaded leaf represents one (disjoint) set, with the set name noted in the box.

given by $T^m(start,start) + T^m(start,escape)$ and $1-T^m(start,crash)$, respectively, where $T(i, j)$ is the probability of a transition from state i to state j.

B. ENCODING OF A UNIFORM-LENGTH INSTRUCTION SET

This appendix contains intermediate derivations for the uniform-length instruction set model.

B.1 Partition Graph

Figure 7 illustrates the partition of the symbols into disjoint sets using the execution model given in Section 4.1.

B.2 Encoding Conventions

The set of branches that are relative to the current instruction pointer with a small offset (defined as being less or equal than 2^{b-1}) is separated from the rest of the branches, because their likelihood of execution is very high. In the analysis we set their execution probability to 1, which is consistent with observed behavior.

A fraction of the conditional branches are artificially separated into L_{MI} and L_{MP} from their original B_{MI} and B_{MP} sets. This fraction corresponds to the probability of taking the branch, which we assume is 0.5. This is similar to the IA32 case, where we assumed that a non-branch-taking instruction could be treated as a linear instruction.

To determine the probability that a symbol falls into one of the partitions, we need to enumerate all symbols in the instruction set. For accounting purposes, when parts of addresses and/or immediate (constant) operands are encoded inside the instruction, each possible instantiation of these data fields is counted as a different instruction. For example, if the instruction "XYZ" has 2 bits specifying one of four registers, we count four different XYZ instructions, one for each register encoding.

B.3 Derivation of the Probability of a Successful Branch (Escape) Out of a Sequence of n Random Bytes

$$\begin{aligned} P(X_n^\star) &= \sum_{i=1,\ldots,n} P(X_i) + P(L)^n \\ &= \sum_{i=1,\ldots,n} P(L)^i P(E) + P(L)^n \\ &= \left(P(E) \sum_{i=1,\ldots,n} P(L)^i \right) + P(L)^n \\ &= P(E)\frac{1 - P(L)^{n+1}}{1 - P(L)} + P(L)^n. \end{aligned} \tag{1}$$

B.4 Derivation of the Lower Limit for the Probability of Escape

$$\begin{aligned} \lim_{n\to\infty} P(X_n^\star) &= \lim_{n\to\infty} P(E)\frac{1 - P(L)^{n+1}}{1 - P(L)} + P(L)^n \\ &= \frac{P(E)}{1 - P(L)}. \end{aligned} \tag{2}$$

REFERENCES

ANDERSON, R. 2003. "Trusted Computing" and competition policy—Issues for computing professionals. *Upgrade IV*, 3 (June), 35–41.

ARBAUGH, W. A. 2002. Improving the TCPA specification. *IEEE Comput. 35*, 8 (Aug.), 77–79.

AVIJIT, K., GUPTA, P., AND GUPTA, D. 2004. Tied, libsafeplus: Tools for dynamic buffer overflow protection. In *Proceeding of the 13th USENIX Security Symposium*. San Diego, CA.

AVIZIENIS, A. 1995. The methodology of N-version programming. In *Software Fault Tolerance*, M. Lyu, Ed. Wiley, New York, 23–46.

AVIZIENIS, A. AND CHEN, L. 1977. On the implementation of N-Version programming for software fault tolerance during execution. In *Proceedings of IEEE COMPSAC 77*. 149–155.

BALA, V., DUESTERWALD, E., AND BANERJIA, S. 2000. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming language design and implementation*. ACM Press, Vancouver, British Columbia, Canada, 1–12.

BARATLOO, A., SINGH, N., AND TSAI, T. 2000. Transparent run-time defense against stack smashing attacks. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00)*, Berkeley, CA. 251–262.

BARRANTES, E. G., ACKLEY, D., FORREST, S., PALMER, T., STEFANOVIC, D., AND ZOVI, D. D. 2003. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, Washington, DC. 272–280.

BEST, R. M. 1979. Microprocessor for executing enciphered programs, U.S. Patent no. 4 168 396.

BEST, R. M. 1980. Preventing software piracy with crypto-microprocessors. In *Proceedings of the IEEE Spring COMPCON '80*, San Francisco, CA. 466–469.

BHATKAR, S., DUVARNEY, D., AND SEKAR, R. 2003. Address obfuscation: An approach to combat buffer overflows, format-string attacks and more. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC. 105–120.

BOYD, S. W. AND KEROMYTIS, A. D. 2004. SQLrand: Preventing SQL injection attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*. Yellow Mountain, China. 292–302.

BRUENING, D., AMARASINGHE, S., AND DUESTERWALD, E. 2001. Design and implementation of a dynamic optimization framework for Windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*.

BUTLER, T. R. 2004. Bochs. http://bochs.sourceforge.net/.

CHEW, M. AND SONG, D. 2002. *Mitigating Buffer Overflows by Operating System Randomization*. Tech. Rep. CMU-CS-02-197, Department of Computer Science, Carnegie Mellon University.

CHIUEH, T. AND HSU, F.-H. 2001. Rad: A compile-time solution to buffer overflow attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS)*, Phoenix, AZ. 409–420.

COHEN, F. 1993. Operating system protection through program evolution. *Computers and Security 12*, 6 (Oct.), 565–584.

CORE SECURITY. 2004. CORE security technologies. http://www1.corest.com/home/home.php.

COWAN, C., BARRINGER, M., BEATTIE, S., AND KROAH-HARTMAN, G. 2001. Format guard: Automatic protection from `printf` format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Washington, DC. 191–199.

COWAN, C., BEATTIE, S., JOHANSEN, J., AND WAGLE, P. 2003. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC. 91–104.

COWAN, C., HINTON, H., PU, C., AND WALPOLE, J. 2000. A cracker patch choice: An analysis of post hoc security techniques. In *National Information Systems Security Conference (NISSC)*, Baltimore MD.

COWAN, C., PU, C., MAIER, D., HINTON, H., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. 1998. Automatic detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX.

COWAN, C., WAGLE, P., PU, C., BEATTIE, S., AND WALPOLE, J. 2000b. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*. 119–129.

DALLAS SEMICONDUCTOR. 1999. DS5002FP secure microprocessor chip. http://pdfserv.maxim-ic.com/en/ds/DS5002FP.pdf.

DOR, N., RODEH, M., AND SAGIV, M. 2003. CSSV: Towards a realistic tool for statically detecting all buffer overflows in c. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. 155–167.

ETOH, H. AND YODA, K. 2000. Protecting from stack-smashing attacks. Web publishing, IBM Research Division, Tokyo Research Laboratory, http://www.trl.ibm.com/projects/security/ssp/main.html. June 19.

ETOH, H. AND YODA, K. 2001. Propolice: Improved stack smashing attack detection. *IPSJ SIGNotes Computer Security (CSEC) 14* (Oct. 26).

FAYOLLE, P.-A. AND GLAUME, V. 2002. A buffer overflow study, attacks & defenses. Web publishing, ENSEIRB, http://www.wntrmute.com/docs/bufferoverflow/report.html.

FORREST, S., SOMAYAJI, A., AND ACKLEY, D. 1997. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*. 67–72.

FRANTZEN, M. AND SHUEY, M. 2001. Stackghost: Hardware facilitated stack protection. In *Proceedings of the 10th USENIX Security Symposium*. Washington, DC.

GERA AND RIQ. 2002. Smashing the stack for fun and profit. *Phrack 59*, 11 (July 28).

HARPER, M. 2002. SQL injection attacks—Are you safe? In Sitepoint, http://www.sitepoint.com/article/794.

IBM. 2003. *PowerPC Microprocessor Family: Programming Environments Manual for 64 and 32-Bit Microprocessors. Version 2.0*. Number order nos. 253665, 253666, 253667, 253668.

INTEL CORPORATION. 2004. The IA-32 Intel Architecture Software Developer's Manual. Number order nos. 253665, 253666, 253667, 253668.

JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M., CHENEY, J., AND WANG, Y. 2002. Cyclone: A safe dialect of c. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA. 275–288.

JONES, R. W. M. AND KELLY, P. H. 1997. Backwards-compatible bounds checking for arrays and pointers in C programs. In *3rd International Workshop on Automated Debugging*. 13–26.

KC, G. S., KEROMYTIS, A. D., AND PREVELAKIS, V. 2003. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*. ACM Press, Washington, DC. 272–280.

KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. 2002. Secure execution via program sheperding. In *Proceeding of the 11th USENIX Security Symposium*, San Francisco, CA.

KLAIBER, A. 2000. The technology behind the crusoe processors. White Paper http://www.transmeta.com/pdf/white_papers/paper_aklaiber_19jan00.pdf. January.

KUHN, M. 1997. *The TrustNo 1 Cryptoprocessor Concept*. Tech. Rep. CS555 Report, Purdue University. April 04.

LAROCHELLE, D. AND EVANS, D. 2001. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Washington, DC. 177–190.

LHEE, K. AND CHAPIN, S. J. 2002. Type-assisted dynamic buffer overflow detection. In *Proceeding of the 11th USENIX Security Symposium*, San Francisco, CA. 81–88.

MILENKOVIĆ, M., MILENCOVIĆ, A., AND JOVANOV, E. 2004. A framework for trusted instruction execution via basic block signature verification. In *Proceedings of the 42nd Annual Southeast Regional Conference (ACM SE'04)*. ACM Press, Huntsville, AL. 191–196.

NAHUM, E. M. 2002. Deconstructing specweb99. In *Proceedings of 7th International Workshop on Web Content Caching and Distribution*, Boulder, CO.

NEBENZAHL, D. AND WOOL, A. 2004. Install-time vaccination of Windows executables to defend against stack smashing attacks. In *Proceedings of the 19th IFIP International Information Security Conference*. Kluwer, Toulouse, France, 225–240.

NECULA, G. C., MCPEAK, S., AND WEIMER, W. 2002. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the Symposium on Principles of Programming Languages*. 128–139.

NERGAL. 2001. The advanced return-into-lib(c) exploits. *Phrack 58*, 4 (Dec.).

NETHERCOTE, N. AND SEWARD, J. 2003. Valgrind: A program supervision framework. In *Electronic Notes in Theoretical Computer Science*, O. Sokolsky and M. Viswanathan, Eds. Vol. 89. Elsevier, Amsterdam.

NEWSHAM, T. 2000. Format string attacks. http://www.securityfocus.com/archive/1/81565.

PAX TEAM. 2003. Documentation for the PaX project. See Homepage of The PaX Team. http://pax.grsecurity.net/docs/index.html.

PRASAD, M. AND CHIUEH, T. 2003. A binary rewriting defense against stack based overflow attacks. In *Proceedings of the USENIX 2003 Annual Technical Conference*, San Antonio, TX.

PU, C., BLACK, A., COWAN, C., AND WALPOLE, J. 1996. A specialization toolkit to increase the diversity of operating systems. In *Proceedings of the 1996 ICMAS Workshop on Immunity-Based Systems*, Nara, Japan.

RANDELL, B. 1975. System structure for software fault tolerance. *IEEE Trans. Software Eng. 1*, 2, 220–232.

RUWASE, O. AND LAM, M. S. 2004. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*.

SCHNEIER, B. 1996. *Applied Cryptography*. Wiley, New York.

SECURITY FOCUS. 2003. CVS directory request double free heap corruption vulnerability. http://www.securityfocus.com/bid/6650.

SEWARD, J. AND NETHERCOTE, N. 2004. Valgrind, an open-source memory debugger for x86-GNU/Linux. http://valgrind.kde.org/.

SIMON, I. 2001. A comparative analysis of methods of defense against buffer overflow attacks. Web publishing, California State University, Hayward, http://www.mcs.csuhayward.edu/simon/security/boflo.html. January 31.

SPEC INC. 1999. *Specweb99*. Tech. Rep. SPECweb99_Design_062999.html, SPEC Inc. June 29.

TCPA 2004. TCPA trusted computing platform alliance. http://www.trustedcomputing.org/home.

TOOL INTERFACE STANDARDS COMMITTEE. 1995. *Executable and Linking Format (ELF)*. Tool Interface Standards Committee.

TSAI, T. AND SINGH, N. 2001. Libsafe 2.0: Detection of format string vulnerability exploits. White Paper Version 3-21-01, Avaya Labs, Avaya Inc. February 6.

TSO, T. 1998. random.C: A strong random number generator. http://www.linuxsecurity.com/ feature_stories/random.c.

VENDICATOR. 2000. StackShield: A stack smashing technique protection tool for Linux. http://angelfire.com/sk/stackshield.

WAGNER, D., FOSTER, J. S., BREWER, E. A., AND AIKEN, A. 2000. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, San Diego, CA. 3–17.

WILANDER, J. AND KAMKAR, M. 2003. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*, San Diego, CA. 149–162.

XU, J., KALBARCZYK, Z., AND IYER, R. K. 2003. Transparent runtime randomization for security. In *Proceeding of the 22nd International Symposium on Reliable Distributed Systems (SRDS'03)*, Florence, Italy. 26–272.

XU, J., KALBARCZYK, Z., PATEL, S., AND IYER, R. K. 2002. Architecture support for defending against buffer overflow attacks. In *2nd Workshop on Evaluating and Architecting System dependabilitY (EASY)*, San Jose, CA. http://www.crhc.uiuc.edu/EASY/.

Received May 2004; revised September 2004; accepted September 2004