Randomized instruction set emulation to disrupt binary code injection attacks

Gabriela Barrantes
David H. Ackley
Trek S. Palmer
Dino Dai Zovi
Stephanie Forrest
Darko Stefanović

Department of Computer Science
University of New Mexico

February 2003

Abstract

Many remote attacks against computer systems inject binary code into the execution path of a running program, gaining control of the program's behavior. If each defended system or program could use a machine instruction set that was both unique and private, such binary code injection attacks would become extremely difficult if not impossible. A binary-to-binary translator provides an economic and flexible implementation path for realizing that idea. As a proof of concept, we describe a *randomized instruction set emulator* (RISE) based on the open-source Valgrind x86-to-x86 binary translator. Although currently very slow and memory-intensive, our prototype RISE can indeed disrupt binary code injection attacks against a program without requiring its recompilation, linking, or access to source code. We describe the RISE implementation, give evidence demonstrating that RISE defeats common attacks, consider consequences of the dense x86 instruction set on the method's effects, and discuss limitations of the RISE prototype as well as design tradeoffs and extensions of the underlying idea.

1 Introduction

Standardizing the interface between software and hardware has proved to be a double-edged sword. On the one hand, it allows independent development of hardware and software, which creates tremendous efficiencies by allowing highly optimized hardware to be combined with flexible software. On the other hand, wide deployment of standardized hard-

ware/software systems has created significant increased risks, by providing a large fixed target for malicious code to attack.

If any flaw in software can be found that allows insertion of foreign information into the execution path of a running machine, that same underlying standardized machine instruction set now becomes a liability, because it allows an attacker to craft a single sequence of code that will have identical deleterious effects on thousands or millions of systems.

If, by contrast, each machine (or program, or process, or other level of execution unit) had a different instruction set, the risks of such code injection attacks would be greatly reduced. In that case, a different code sequence would be required for each system attacked. To the degree that the number of possible instruction sets is large and to the degree that it can be made difficult to determine, from the outside, what specific instruction set any particular machine or program is using, the cost of developing an attack against that machine rises as well, and the cost of attacking multiple machines becomes linear rather than constant in the number of machines.

Is it possible to achieve the benefits of standardization without continuing to accept the increased epidemic risks of the widespread computing monoculture we have created? In this paper we answer that question in the affirmative, demonstrating one way to do so via *randomized instruction set emulation* (RISE), a technique that deliberately obscures the standardized machine instruction set using a private randomized scrambling mechanism. With RISE, even if a software flaw allows machine code injection, the attacker will also require hard-to-obtain information specific to the machine in order to craft a successful binary code attack. The basic perspective underlying RISE is that when an attempt to exploit a software flaw occurs, it is far better for the victim program to crash or even behave randomly than to yield control to the attacker.

The scrambling function is designed so that it is infeasible to create code sequences to perform a desired function (e.g., an attack) without access to a long secret key that is unique to each program execution. However, with knowledge of the scrambling mechanism and the key, code designed for the unique randomized machine can easily be transformed back into code for the standardized physical hardware.

In this paper we provide a proof-of-concept RISE system, building randomized instruction set support into a version of the Valgrind x86-to-x86 binary translator [23]. We adopt an attack model focusing on network code injection into running programs and assume that the contents of local disks are trustworthy before the attack has occurred. Given that model, we describe a *randomizing loader* for Valgrind that randomly scrambles code sequences loaded into emulator memory from the local disk using a hidden key. Then during Valgrind's emulated instruction fetch cycle, we unscramble the fetched instruction, yielding unaltered x86 machine code runnable on the physical machine.

When binary attack code, arriving over the network, exploits a bug and manages to interpose itself into the emulator execution path, the injected code will not have been scrambled by the loader. Consequently, when the attack code is fetched and unscrambled by the emulated instruction unit, it will appear as an essentially random string of bits. Despite

the density of the x86 instruction set, our initial studies suggest that most random code sequences will encounter an address fault or illegal instruction quickly, aborting the program. Thus with RISE, an attack that would otherwise take control of a program is downgraded into a denial-of-service attack against the exploitable program. Regardless of what flaw is exploited in a protected program—whether well-known or entirely novel—the network binary code injection attack will fail with very high probability.

2 Background and 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 [4, 21, 5], and, more recently, has been proposed as a method for improving security [8, 14, 9]. At least two other (nondiversifying) approaches have been developed for protecting against stack-smashing attacks—the primary example of the threat model we have adopted. The first of these involves inserting additional controls and explicit checking for pre- and post-conditions. The second approach relies on special hardware support and/or kernel modifications in combination with additional checks. In the following three subsections we discuss related work in automated diversity as well as these two different approaches to protecting against stack-smashing attacks. Our diversification approach should be viewed as complementary to these other methods.

2.1 Automated diversity and arms races

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, say around ten.

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 manually rewrite the attack code for each new computer that is attacked. Here 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 infeasible here, and the diversity must be produced auomatically.

Cowan et al. introduced a classification of diversity methods applied to security (called "security adaptations") which classifies adaptations based on *what* is being adapted: either the interface or the implementation [9]. Interface adaptations modify code layout or access controls to interfaces, without changing the underlying implementation to which the interface gives access. Implementation adaptations, on the other hand, do modify the underlying

implementation of some portion of the system to make it resistant to attacks. RISE can be viewed as an interface randomization at the machine code level.

Earlier work in automated diversity for security has experimented with diversifying data layouts [8, 20], file systems [9], and system-call interfaces [24]. In addition, several projects address our code-injection threat model directly, and we now describe those projects briefly.

In 1997, Forrest et al. presented a general view of the possibilities of diversity for security [14], introducing the idea of deliberately diversifying data layouts as well as code, and demonstrated an example of diversification that randomly padded stack frames so that exact return address locations would be less predictable, making it harder for an attacker to locate the return address and other key stack offsets. Developers of buffer overflow attacks have developed a variety of workarounds—such as "ramps" and "landing zones" of no-ops and multiple return addresses—aimed at coping with variations across different versions or different compilations of the vulnerable software. Deliberate diversification via random stack padding coerces an attacker to use such generalization techniques; it also necessitates larger and larger attack codes in proportion to the size range of random padding employed.

The StackGuard system [10] provides a counter-defense against landing zones and similar attack techniques by interposing a hard-to-guess "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 almost surely change the canary value and thus be detected.

Such examples illustrate some general points about automated diversity. First, the diversifications will be most successful if they are designed in such a way that a knowledgable adversary cannot easily create an automated method to overcome the diversification. In many cases, this consideration will force the diversifier to use secondary information hiding and obfuscation methods in order to protect the diversification. A second consideration, which applies to many areas of security, including diversity, is that a defense or an attack does not have to be 100% effective to be useful, and that "arms races" between attackers and defenders are common. Random stack padding defeats the most specific attacks cheaply and increases the need for more generalized attack techniques; a technique such as StackGuard counters more sophisticated stack-smashing attacks but requires more overhead, and so forth.

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's hard to get much done when you don't know the language, RISE is another technique in the defender's arsenal against binary code injection attacks.

2.2 Enforcing security with optimizing interpreters

It has been noted that the current trend in binary-to-binary optimizing interpreters could be used for more detailed inspection of executing code, because every control transfer is detected during the interpretation process. Kiriansky et al. [16] proposed a method called "code shepherding" in which various policies are defined to govern allowable control transfers. Two of those types of policies—those based on code origins, and those based on restricted control transfers—are particularly relevant to the RISE approach.

Code origins 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 code policy is the code deposited in the stack by signals, which is handled specially by Valgrind.

Another relevant policy type in the code shepherding approach is *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, instead of trying to detect and block all unintended control transfers, we let them happen and rely on the RISE 'language barrier' to ensure the injected code will fail. Note that the possibility of an unintended transfer to *existing* code—rather than to injected code—is outside the threat model that RISE most naturally addresses, and there is thus a potential for loss of control that needs to be blocked via other mechanisms (see Sec. 5).

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. In general, although divorcing policy from mechanism is a valid design principle, when it comes to security it is too easy to make mistakes in defining policies, and a mechanism that inherently enforces a correct policy is preferable. We are currently developing a randomized version of control transfers to expand the threat model to cover unintended transfers to existing code. Combinations of RISE and code shepherding techniques could also be considered.

2.3 Other approaches to dealing with buffer overflows

In addition to the stack-frame padding and canary methods described earlier, several other solutions have been proposed to deal specifically with buffer overflows [11]. These solutions employ compiler extensions, hardware characteristics, kernel modifications, library modifications, or static analysis to prevent and detect exploitation of buffer overflow vulnerabilities.

If the source code to a security-critical application is available, compiler extensions

can be used to instrument the executable with additional controls and explicit checking. For example, both Stack Shield [26] and StackGuard [10] instrument subroutine prologues to check the integrity of the return address in the subroutine linkage before transferring control to that location. Stack Shield does this by ensuring that the return address is within the process text segment or by matching the return address with a copy kept in an out-of-band return address stack in the data segment. These methods protect against basic stack smashing buffer overflows, but they do not protect against all code injection attacks, and exploitation via stack overflows is still possible [22].

When application source code is not available, modifications to the operating system kernel or shared libraries can provide system-wide buffer overflow defense mechanisms by taking advantage of specific hardware characteristics. For example, StackGhost [15] makes clever use of register windows on the SPARC architecture, so that if the return address is overwritten by an attacker while it is stored on the stack, the resulting value will not be valid. Similarly, the PaX Linux kernel patch relies on a property of newer x86 implementations (dual translation lookaside buffers) to implement non-executable memory pages on the Intel architecture. Although these solutions have the advantage of not forcing recompilation of user applications, they are by definition architecture-specific and non-portable. In addition, StackGhost does not protect all control transfers, only stack return addresses, so other exploit vectors may still be possible.

A nonexecutable stack is also implemented by Solar Designer's Openwall Linux kernel non-executable stack patch [12]. Solar Designer's system modifies the kernel to ensure that control is never returned from the kernel to user code executing in the stack segment. Although non-executable stacks (and the more general PaX non-executable memory pages) do prevent injected code from executing on the stack, a clever attacker can still bypass non-executable segments through *return-into-libc* exploits [17].

Yet another approach modifies the loading process to change the characteristics of the process segments. In particular, PaX [19] randomizes the address space layout, loading libraries at random addresses in the process address space, which makes it difficult for exploits to jump directly to the address of a needed function. Openwall changes the default loading addresses of the segments, making it almost impossible for injected code to specify its absolute virtual address. Although this approach is close to ours because it protects individual processes, it focuses on the address space rather than the code.

Finally, there are other static solutions, such as library call wrappers [1] and code examination to check for known error conditions [27]. Wrappers around library calls have the limitation of not necessarily covering all libraries or all calls, and the overflow can happen in the main program code instead of the library. Also, this method can incur signficant execution time penalties. Code examination tools are useful, primarily to program designers, but they are less practical for the end-user who wants to run an application in a secure way.

3 Technical Approach and Implementation

In this section we describe our prototype implementation of RISE using Valgrind [23] for the Intel x86 architecture. The RISE strategy is to design a system that provides 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 our implementation.

3.1 Machine abstraction level

Not all levels of machine abstraction are equally amenable to a randomization strategy. We identified the following requirements for a computational level to be a promising candidate:
Risk of attack: At least one of the source languages that are compiled to the machine language must be vulnerable to programming errors that can produce undesirable machine behavior. C provides an example of a commonly used source language that has many constructs that make it prone to subtle errors that give an attacker access to private areas of memory. If such vulnerabilities did not exist, then there would be no point in protecting against them.
Clear trust boundary: Local and trusted programs must be easily identifiable. For example, we assume in our experimental setup that program files on local disks can be trusted.
Majority of trusted code: The set of local (and trusted) programs must constitute a significant proportion of the executing programs on the machine. If we choose a machine which runs mostly external code, then we will need a different policy for distinguishing trusted from untrusted code. For example, randomizing Javascript in web browsers would fare poorly under our scheme because most of the code the web browser runs is external. Even though external Javascript can be a source of dangerous code, its identification as trusted or untrusted must be made according to signatures, origination address, or a similar authentication mechanism, in which case the RISE approach becomes superfluous.

For these reasons, we selected the native instruction set that runs in x86 processors. Machine language (or C) is certainly a risky language as described above; local programs are easily identifiable; and new code is presumably well-authenticated before it is installed for general use. Clearly, the set of local trusted programs is much larger than the set of "import-on-the-fly" programs.

Compared to higher-level virtual machines, however, native instruction sets have the drawback that they are most often physically encoded and not modifiable. Thus, we chose to work at an intermediate level with interpreters that perform binary-to-binary translation. Several such tools have been developed [6, 7], and their reported slowdown is very small; indeed, they occasionally improve the performance of programs. However, most dynamic optimizers are not open source, and the proposed randomization requires access to the source code of the interpreter itself. To overcome this obstacle, we decided to use Valgrind [23]

for our prototype implementation. We are currently developing our own binary-to-binary translator and plan to port RISE to it when it is mature.

Valgrind was designed to be a tool for detecting memory leaks, but it contains a complete binary-to-binary translator. Because of the extensive memory access checking performed by Valgrind, it is slow. However, as our intention is to present a proof of principle, correctable inefficiencies of the interpreter are tolerable. As we discuss in Section 4, the time and space costs of adding RISE to Valgrind are generally insignificant. Though scrambling is performed byte-by-byte, it is only done once, at load time. Descrambled code sequences are kept in a suitably protected cache and run at full processor speed.

3.2 Byte-level encoding

There are at least two basic approaches to creating private new instruction sets: (1) modifying the operation codes and layouts in the virtual machine itself, or (2) encrypting the machine code and then decrypting it before it is executed by the virtual machine. The first option would require a new virtual machine for every randomized program, or alternatively, all the programs on a single machine would have to share the virtual machine. The second option allows much greater flexibility, and makes it possible to load ELF files directly, without either recompiling or recalculating offsets for each new language instruction set.

We chose an extremely straightforward encryption scheme: Select a key of length n bytes, where n is a parameter of the system; XOR the key with the first n bytes of the machine code, and repeat the operation with the same key until the executable is scrambled. The key is generated randomly for every new process. When decoding, the byte to be decoded is XOR-ed with its corresponding part (subkey) of the key. The subkey index in the key is easily recovered from the instruction pointer (EIP) by the following operation: (EIP mod n). This allows memory that was encoded linearly to be decoded correctly regardless of the order of instruction execution, even though x86 instructions have varying lengths. It also makes our method less likely to be circumvented as decoding happens at the byte fetch level.

3.3 Load time randomization of code segments

There are also choices about *when* to recode the executable. Scrambling ELF files on disk has liabilities ranging from user inconvenience to increased risk of key discovery to problems dealing with dynamic libraries. For such reasons, we decided to recode the executable at load time. This allows us to create new keys for each execution, and solves the library problem; because we are writing to memory that belongs to shared objects, which are copy-on-write, a private copy of the code we encode is stored in the virtual memory of the process. This is a one-time cost and will have a very small effect for most processes. We still have the benefit of not having the plaintext code in memory, but we pay the price of using additional memory (defeating the purpose of shared libraries).

3.4 Decoding at fetch time

An emulator such as Valgrind simulates the operation of the CPU and goes through a fetch cycle. We interpose at the point where the next byte(s) are read from program memory, and decode them before passing the decoded result to the emulator. The code in memory is never modified. Eventually, however, an instrumented (or optimized), semantically-equivalent version of the basic block read is written to the cache. We do not interfere with the generation of the cached code, which is the only plaintext version of the process code. The next subsection describes how we protect the cache.

3.5 Protecting Interpreter Data

Translation is slow. Completely separating the interpreter's address space and the address space of the target process slows down data accesses to an extreme degree, and this problem is exacerbated when the interpreter uses a native code cache to speed up execution. For efficiency, the RISE interpreter is best located in the same address space as the target binary, but of course this introduces some security concerns. If an attacker is aware that the vulnerable program is being run under RISE, the attacker could attempt to design an attack that overwrites RISE data (rather than process data). This is a problem because RISE's own internal code cannot be encrypted. Therefore we must protect RISE's code and any code located in the code cache (often stored in the heap). We write-protect the cache whenever it executes outside the interpreter, and we carefully control how the interpreter inserts new code fragments into the cache. Because RISE will inspect all code executed by the target program, RISE can catch any attempts by the program to manipulate the protection of RISE structures. This simple check adds little to the overhead of interpreting a system call.

3.6 Implementation issues

The major unexpected problem that we encountered during the implementation was that the boundary between Valgrind and the process Valgrind emulates is not strictly enforced. We found 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 never-descrambled instructions, causing Valgrind to abort.

In a sense, it is a testament to the robustness of the RISE approach that these latent 'boundary crossings' were made so immediately apparent. Although references to most common libc functions were eliminated in the Valgrind implementation, a few symbols referring to very low-level compiler-version or system-specific functions remained. We worked around these dangling unresolved references by creating inline function definitions in the Valgrind binary. Although not an ideal solution, this quick fix allowed us to develop a working implementation of RISE within Valgrind.

These dangling-reference problems are endemic in binary translation work, and creating a self-contained interpreter is a highly non-trivial task. One of the benefits of the SIND [18]

dynamic binary translation system (currently under development), is that it will be wholly self-contained precisely to support applications such as RISE.

4 Experimental Results

Our prototype RISE implementation has only recently begun working and so our data is still thin. However, we can report early successes both on running programs successfully under normal conditions and on disrupting machine code injection attacks. As sample programs, we have run both `ls`—a relatively small program—and `Apache`—a somewhat larger one—under RISE without problems. At the expense of substantially increased space and time requirements (discussed below), useful programs can be protected with RISE today.

We have also tested RISE against two synthetic attacks and a vulnerability test for the Apache chunk encoding [3]. The synthetic attacks we use, published in [13], create a vulnerable buffer—in one case on the heap and in the other case on the stack—and inject shellcode into it. Without RISE, both attacks successfully spawn a root shell (when run by a root process). On the other hand, with RISE, both attacks are successfully stopped—the system detects an invalid x86 operation and aborts, printing the address of the offending instruction (in these two attacks, a heap address and a stack address, respectively). The attack code was descrambled without ever having been scrambled, turning it effectively into random bits.

These synthetic attacks provide baseline data showing that the RISE approach—that of guarding the code itself, rather than guarding the access to it—can stop some of the same sorts of attacks that existing buffer overflow solutions address.

The vulnerability test for the chunk encoding vulnerability [25] was run on an unpatched Apache version 1.3.12. As expected, it segfaults when run directly—since that is what the vulnerability test is designed to do. When run under RISE, rather than segfaulting it reports an illegal memory access, owing to the scrambling. We are in the process of modifying the vulnerability test to inject some code (without attempting to gain access), to show that RISE operates just as well in large applications.

These early data only give confidence that we successfully implemented what we knew had to work from first principles: That randomizing the instruction set would disrupt machine code injections unaware of the effective instruction set. More important is attempting to develop and defeat attacks given awareness of the RISE strategy. We discuss the 'unintended transfer to existing code' problem in Sections 2 and 5.

4.1 What happens when an attack fails

There is also a question about how safe it is to be executing effectively random bits.

Suppose a software flaw is exploited and foreign machine code is injected into the execution path of a RISE protected program. Since the foreign code was created without

access to the program's hidden random key, when the foreign code is unscrambled for execution it will be essentially random bit values and will not perform any specific function on behalf of the attacker. But if such random code does not do the attacker's bidding, what does it do? The expectation is that such random code strings will be likely to cause the attacked program to crash quickly—but is that actually true?

Our prototype RISE produces randomized instruction sets that are in a byte-for-byte correspondence with actual x86 instructions. This has the significant advantage that the transformation process doesn't affect either the code size or the layout, allowing us to perform the randomization task more simply and very late in the game, at load time. However, this size correspondence also has a potential drawback, in that so much of the x86 opcode space has been defined that executing a random byte might be quite likely to do *something*, rather than just being an illegal instruction. Just how far should we expect an x86 processor to get through a random byte sequence before dying?

To get a feel for the likelihoods involved, we performed the following test: We built a small test program that contained a rootshell exploit coded into it in x86 machine code (specifically, the shellcode from '`testsc2.c`' in [2]). When the program ran, it first randomized the exploit code in place using a random number seed supplied on the command line, and then it transferred control to the beginning of the scrambled sequence via a function call. We ran this test program $10,000$ times varying only the random seed. Table 1 summarizes the results.

	Count	*Percent*	*Cumulative*
Signalled	9783	97.83%	97.83%
Returned	188	1.88%	99.71%
Looped(timeout)	29	0.29%	100.0%
Acquired shell	0	0%	0.0%
Total tests	10000		

Table 1: Outcomes of executing randomized shell acquisition code. Distribution of signals was SIGSEGV=$8,721$, SIGILL=$1,010$, SIGBUS=37, SIGFPE =15.

In this experiment, about 97.8% of randomizations lead to the program aborting by one of four signals. SIGILL is an illegal instruction, SIGFPE is a floating point exception (such as division by zero), and SIGSEGV and SIGBUS are two varieties of bad addressing problems. Of the remaining cases, somewhat less than 2% of the time the randomized code managed to effect a return to the caller, and under one third of a percent of cases lead to the program entering an (apparently) infinite loop. In none of the $10,000$ test cases, of course, did the attack code succeed in accessing the command interpreter `/bin/sh` as it was designed to do.

There are caveats to this data. Note that SIGSEGV's are by far the most commonly emitted signal—but that could be misleading since the test program is so small. A larger

program would have a correspondingly larger space of legal addresses and would expect to generate fewer SEGV's. Also, interpreting the 2% of 'returned' cases are somewhat problematic. For one thing, attack code is often 'returned into' rather than being called directly so returning in this way wouldn't be an option; since our test driver does not actually smash its own stack the possibility of a clean return is likely much higher here than during an actual attack. (We are in the process of developing better tools for estimating these probabilities). In addition, although the code appeared to return successfully, it is also possible that some machine state may have been randomly altered before the return in ways that would lead to a crash sometime later.

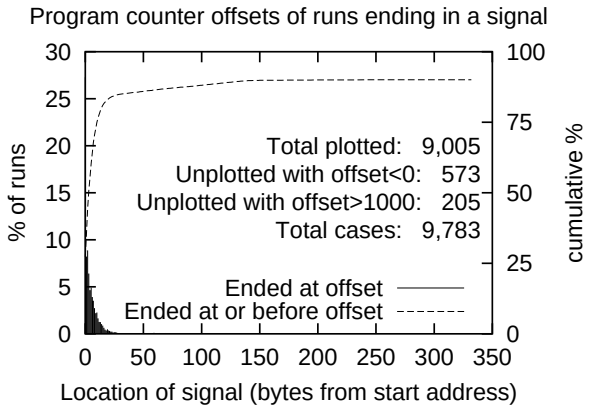

Figure 1: Distribution of signal locations relative to the beginning of the randomized attack code.

Nonetheless, this case study is heartening in the sense that the vast majority of randomizations of a genuine attack do indeed simply cause a program crash. Although our preliminary data does not directly answer the question of *how fast* the crashes tend to occur, Figure 1 provides some encouraging indirect data on that point, illustrating where the program counter was when the signal occurred in the $9,783$ cases that lead to a signal. There is a strong peak at 0—in over one quarter of all test cases, when the program was stopped it was at the very beginning of the randomized attack code,[1] and the fraction of attacks fall off rapidly away at increasing offsets. Due to random control transfers, we also observed signals from locations outside the randomized attack window a total of 7.78% of the time (573 cases beyond the end of the window and 205 before the beginning).

More thorough and theoretically pleasing, if perhaps more invasive as well, RISE architectures could avoid worrying about this problem by mapping to a larger instruction set, which can be tuned in size so that whatever percentage desired of incorrect unscramblings

[1] Since we did not actually count instructions executed, we cannot completely rule out the possibility that a random code sequence may actually have executed for a time, modifying itself, and then returned to where it started and died.

will likely lead immediately to an illegal instruction.

4.2 Performance

Although we are willing to pay the heavy cost introduced by the memory checking engine of Valgrind in order to build our proof of concept, it is possible to get an approximation to the overhead of RISE over that of Valgrind.

Table 2 shows the cost in time and memory space imposed by Valgrind alone and RISE against the baseline native application for an Apache server. The time measured is the cumulative time for 10 GET operations of the main index page.

	Native	Valgrind	RISE
space (KB)	1008	11872	12060
time (s)	0.010	0.850	0.890

Table 2: Space and time overhead of Valgrind and RISE over native execution of the Apache server.

5 Discussion

Randomized instruction sets prevent dynamically injected machine code from being executed, as do non-executable stack and heap techniques. And, RISE shares many of the advantages of these earlier projects, including the ability to randomize ordinary executable files and no special compilation requirements. Our approach differs, however, from non-executable stacks and heaps in important ways. First, most non-executable stack/heap systems (such as PaX) are applied systemwide, while RISE can be selectively employed on a per-process basis. This distinction becomes important, for example, when we consider Java Virtual Machines, where a runtime compilation process generates code, places it on the heap, then later jumps to it. In a system with a traditional non-executable heap, JVMs cannot run at all. In RISE, however, the JVM process can simply be run outside of RISE without compromising the security of other running processes. Second, enabling non-executable stack/heap protection on a system often requires additional hardware or operating system modification. RISE, however, is a user-level application, and requires no special hardware or OS changes. RISE is capable of running on any binary-to-binary translator, and so can run on any system with such software. As implemented, RISE can be seamlessly deployed on stock Linux distributions by any user capable of expanding a tarball.

Because RISE uses simple runtime encryption to achieve security, a naive observer might ask: How is this different from TCPA? While the published TCPA trusted computing system places encrypted code in memory and decrypts it on the fly, the TCPA system does much more. One of the main focuses of the TCPA is to facilitate Digital Rights Management

(DRM), and the TCPA system contains an extensive document examination and revocation system. RISE is neutral about digital rights and aims only to increase the runtime security of software. Consequently, RISE contains none of the extensive machinery found in TCPA for supporting DRM. RISE is also meant to be fully user-controllable, and so suffers from none of the control issues raised by TCPA. For instance, a RISE installation is answerable only to the user who is using it. No outside authority can alter RISE's behavior without user consent.

In its current state, RISE does not explicitly protect against *return-into-libc* techniques [17, 12] in which a code pointer is rewritten to point to code already loaded into the process address space, a technique commonly employed to evade non-executable stack and heap segments. Return-into-libc exploits could be prevented by loading `libc` at a low memory address or by randomizing the load addresses of all executable memory regions in the process. A system using RISE could also implement a randomized address space to combat exploits that reuse already loaded code. It is also possible that RISE by itself would increase the difficulty of return-into-libc attacks by preventing the attacker from 'droping down to machine code' to execute bits of glue code putting the attack together.

There are other important examples of "code-reuse" attacks, such as the format string vulnerability, in which a series of directives is inserted into the format string to print out successive words from the stack, revealing variable data and subroutine return addresses. This gives the attacker the dynamic location of a known instruction, the subroutine call to the vulnerable function. A second example is the recent buffer overflow in the Solaris `login` program, which was exploited by overwriting a flag variable that indicated whether or not the user needed to be authenticated. A system implementing randomized address spaces by randomizing load addresses could be successfully exploited in some of these situations. A more thorough randomization of the address space, including instruction and procedure reordering may be called for, and this can easily be accomplished within RISE. Although neither the system described in this paper nor any other published techniques, address "code-reuse" attacks, randomized instruction sets with address space randomization would stop all code injection attack techniques that have so far been demonstrated "in the wild." And, they would do so efficiently and conveniently, without specialized hardware or OS support, leaving the user in control.

Although Valgrind has some limitations, discussed in Section 3, we are optimistic that improved designs and implementations of "randomized machines" would vastly increase performance and reduce resource requirements, and potentially expand the range of attacks the approach can mitigate. Aside from performance issues, the greatest weakness in the current implementation arises from the dense packing of legal x86 instructions in the space of all possible byte patterns. A random scrambling of bits is likely to produce a different legal instruction, which is executable. Even doubling the size of the encoding (in bits) for each instruction would enormously reduce the risk of a processor successfully executing a long enough sequence of undescrambled instructions to do damage. Although our preliminary analysis shows that this risk is very low, even with the current implementation, we

believe that emerging architectures, similar to Crusoe, will make it possible to reduce the risk even further.

6 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 to exhibit common injection techniques. Although our main goal in developing RISE was to enhance security, we note that there are other potential advantages to our randomization, such as additional protection for proprietary code while it is resident in memory.

The current RISE implementation does not offer 100% protection against code-injection attacks. As mentioned earlier, there is a small probability that any given undescrambled binary code sequence could be executed for at least a few instructions. Even if it were not executing with the intended semantics of its author, there is some risk that the random instruction sequence could still do damage, most likely causing a process to die. We argue that even in this case, RISE provides an important advantage, because it turns a focused attack into a form of denial-of-service. Certainly, it is desirable to have provable absolute methods of defense, and it's conceivable that a future implementation of RISE might provide the same kind of guarantee that modern cryptographic methods provide. But, that level of certainty is rare in systems of nontrivial complexity. As we have argued in the past [14], inexpensive and imperfect methods which raise the security arms race to a new level make important contributions to the security of our computing infrastructure.

Although our paper illustrates the idea of randomizing instruction sets at the machine code level, the basic concept should be applicable 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 else no externally useful computation can be performed. However, as the recent SQL attacks illustrate, it is increasingly dangerous to express running programs in externally-known languages. Randomized instruction set emulators are one step towards reducing that risk. More generally, the technique presented here illustrates the importance of developing unique properties for each program copy, i.e., the importance of diversity, such that widely replicated attack methods are unlikely to succeed.

Acknowledgments

The authors gratefully acknowledge the partial support of the National Science Foundation (grants ANIR-9986555, CCR-0219587, and CCR-0085792), 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, Microsoft Research, Intel Corporation, and the Santa Fe Institute.

References

[1] libsafe - Detect and handle buffer overflow attacks. In *http://www.gnu.org/directory/security/net/libsafe.html*.

[2] ALEPH ONE. Smashing the stack for fun and profit. *Phrack 49*, 7 (Nov. 1996).

[3] APACHE HTTP SERVER PROJECT. Apache Web Server Chunk Handling Vulnerability. No. CAN-2002-0392, CERT VU#944335, CERT advisory CA-2002-17.

[4] AVIZIENIS, A. The Methodology of N-Version Programming. In *Software Fault Tolerance* (1995), M. Lyu, Ed., John Wiley & Sons Ltd., pp. 23–46.

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

[6] BALA, V., DUESTERWALD, E., AND BANERJIA, S. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation* (Vancouver, British Columbia, Canada, 2000), ACM Press, pp. 1–12.

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

[8] COHEN, F. Operating System Protection through Program Evolution. *Computers and Security 12*, 6 (Oct. 1993), 565–584.

[9] COWAN, C., HINTON, H., PU, C., AND WALPOLE, J. A Cracker Patch Choice: An analysis of Post Hoc Security Techniques. In *National Information Systems Security Conference (NISSC)* (Baltimore MD, October 16-19 2000).

[10] COWAN, C., PU, C., MAIER, D., HINTON, H., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. Automatic Detection and Prevention of Buffer-Overflow Attacks. In *7th USENIX Security Symposium* (San Antonio, Texas, Jan. 1998).

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

[12] DESIGNER, S. Non-executable user stack. In *http://www.openwall.com/linux*.

[13] FAYOLLE, P., AND GLAUME, V. A buffer overflow study, attacks & defenses. In *http://www.enseirb.fr/ glaume/indexen.html*.

[14] FORREST, S., SOMAYAJI, A., AND ACKLEY, D. Building Diverse Computer Systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems* (1997), pp. 67–72.

[15] FRANTZEN, M., AND SHUEY, M. Stackghost: Hardware facilitated stack protection. In *10th USENIX Security Symposium* (Washington D.C., August 2001).

[16] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. Secure Execution Via Program Sheperding. In *Proceeding of the 11th USENIX Security Symposium* (San Francisco, California, August 2002).

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

[18] PALMER, T., ZOVI, D. D., AND STEFANOVIC, D. SIND: A framework for binary translation. Tech. Rep. TR-CS-2001-38, Department of Computer Science, University of New Mexico, Dec. 2001.

[19] PAX TEAM. Non executable data pages. In *http://pageexec.virtualave.net/ pageexec.txt* (2002).

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

[21] RANDELL, B. System Structure for Software Fault Tolerance. *IEEE Transactions in Software Engineering 1*, 2 (1975), 220–232.

[22] RICHARTE, G. Four different tricks to bypass stackshield and stackguard protection. http://www.corest.com/files/files/11/StackguardPaper.pdf, June 2002.

[23] SEWARD, J. Valgrind, an open-source memory debugger for x86-GNU/Linux. In *http://developer.kde.org/ sewardj/* (2002).

[24] SONG, D. Unpublished Report, 1999.

[25] TESTA, J. Re: ISS advisory: Remote compromise vulnerability in Apache HTTP server. In *http://online.securityfocus.com/archive/1/277738/2002-06-16/2002-06-22/0*.

[26] VENDICATOR. StackShield: A stack smashing technique protection tool for Linux. In *http://angelfire.com/sk/stackshield*.

[27] WAGNER, D., FOSTER, J. S., BREWER, E. A., AND AIKEN, A. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Network and Distributed System Security Symposium* (San Diego, CA, february 2000), pp. 3–17.