On the General Applicability of Instruction-Set Randomization

Stephen W. Boyd, Gaurav S. Kc, Michael E. Locasto, Angelos D. Keromytis, and Vassilis Prevelakis

Abstract—We describe Instruction-Set Randomization (ISR), a general approach for safeguarding systems against *any* type of code-injection attack. We apply Kerckhoffs' principle to create OS process-specific randomized instruction sets (e.g., machine instructions) of the system executing potentially vulnerable software. An attacker who does not know the key to the randomization algorithm will inject code that is invalid for that (randomized) environment, causing a runtime exception. Our approach is applicable to machine-language programs and scripting and interpreted languages. We discuss three approaches (protection for Intel $x86$ executables, Perl scripts, and SQL queries), one from each of the above categories. Our goal is to demonstrate the generality and applicability of ISR as a protection mechanism. Our emulator-based prototype demonstrates the feasibility ISR for $x86$ executables and should be directly usable on a suitably modified processor. We demonstrate how to mitigate the significant performance impact of emulation-based ISR by using several heuristics to limit the scope of randomized (and interpreted) execution to sections of code that may be more susceptible to exploitation. The SQL prototype consists of an SQL query-randomizing proxy that protects against SQL injection attacks with no changes to database servers, minor changes to CGI scripts, and with negligible performance overhead. Similarly, the performance penalty of a randomized Perl interpreter is minimal. Where the performance impact of our proposed approach is acceptable (i.e., in an already-emulated environment, in the presence of programmable or specialized hardware, or in interpreted languages), it can serve as a broad protection mechanism and complement other security mechanisms.

Index Terms—Interpreters, emulators, buffer overflows, SQL injection, randomization, security, performance.

1 Introduction

Software vulnerabilities have been the cause of many computer security incidents. Among these, *code-injection attacks* are perhaps the most widely exploited type of vulnerability, accounting for a large percentage of the CERT advisories in the past few years [41]. One of the most traditional vectors of code injection, buffer overflow attacks, typically [32] (but not always [13]) exploit weaknesses in software that allow them to alter the execution flow of a program and cause arbitrary code to execute. This code is usually inserted in the targeted program, as part of the attack, and allows the attacker to subsume the privileges of the program under attack. Because such attacks can be launched over a network, they are regularly used to break into hosts or as an infection vector for computer worms. In their simplest (and perhaps most common) form [2], [32], such attacks overflow a buffer in the program stack and cause control to be transferred to the injected code. Similar attacks overflow buffers in the program heap [27] or use other injection vectors (e.g., format strings).

However, such code-injection attacks are by no means restricted to languages like C; attackers have exploited failures in input validation of Web CGI scripts to permit them to execute arbitrary SQL [11], Unix command line (shell) [10], and Perl instructions on the target system, respectively. Similar attacks have been demonstrated and successfully exploited by self-propagating malcode against websites using PHP. The intuition behind the latter type of attacks is that logical expressions within a predefined query can be altered simply by injecting language operations of the attacker's choice. This injection (which is quite effective in practice [3], [25]) typically occurs through a Web form; the associated CGI script neglects to perform appropriate input validation. One solution to the problem is to improve programming techniques or to use type-safe languages. For example, improved SQL programming practices include escaping single quotes, limiting input length, and filtering exception messages. Despite these suggestions, vulnerabilities continue to surface, implying the need for a different and systematic approach.

Although the specific techniques used in each attack differ, they all result in attackers executing code of their choice. This capability implies that attackers know what "type" of code (e.g., $x86$ code, SQL, Unix shell commands) can be injected.

1.1 Our Approach

This observation led us [21] (and concurrently others [4], [5]) to introduce a general approach for preventing code-injection attacks, *instruction-set randomization (ISR).* By randomizing the underlying system's instructions, "foreign" code introduced by an attack would fail to execute correctly, regardless

- *S.W. Boyd is with the Fraud Management Division, SAS Institute Inc., One PPG Place, Suite 2950, Pittsburgh, PA 15222. E-mail: Stephen.Boyd@sas.com.*
- *G.S. Kc is with Google Inc., 76 9th Avenue, New York, NY 10011. E-mail: gskc@google.com.*
- *M.E. Locasto is with the Department of Computer Science, George Mason University (GMU), Science and Technology II, Rm. 333, 4400 University Drive, MS 4A4, Fairfax, VA 22030. E-mail: mlocasto@gmu.edu.*
- *A.D. Keromytis is with the Department of Computer Science, Columbia University, M.C. 0401, 1214 Amsterdam Avenue, New York, NY 10027. E-mail: angelos@cs.columbia.edu.*
- *V. Prevelakis is with the AEGIS Research Center in Information Security, 2 Alkimachou Street, 116 34 Athens, Greece. E-mail: vpmmviii@prevelakis.net.*

Manuscript received 28 Aug. 2005; revised 26 Sept. 2006; accepted 26 Sept. 2008; published online 7 Oct. 2008.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-0116-0805.
Digital Object Identifier no. 10.1109/TDSC.2008.58.

1545-5971/09/$25.00 © 2009 IEEE Published by the IEEE Computer Society

of the injection approach. *Thus, our approach addresses (at least in principle) not only stack- and heap-based buffer overflow attacks but* **any** *type of remote code-injection attack.* What constitutes the instruction set to be randomized depends on the system under consideration: common stack- or heap-based buffer overflow attacks typically inject machine code that corresponds to the underlying processor (e.g., Intel $x86$ instructions). For Perl injection attacks, the "instruction set" is the Perl language, since any injected code will be executed by the Perl interpreter. To simplify the discussion, we will initially focus on machine-code randomization in our discussion of ISR, although we discuss our prototype randomized Perl in Section 3.

Randomizing an arbitrary instruction set, such as $x86$ machine code, involves three components: the randomizing element, the execution environment, and the loader. Where the processor supports such functionality (e.g., the TransMeta Crusoe, some ARM-based systems, or other programmable processors or interfaces between main memory and processor [34]), our approach should be implementable without noticeable loss of performance, since the randomization process is straightforward, as we report in Section 2. We describe the necessary modifications to the operating system and the randomizing element. We use a modified version of the *bochs-$x86$ Pentium* emulator to determine the feasibility of our design and to identify the necessary changes to the operating system and user-space tools. Generally, the loss of performance associated with an emulator is unacceptable for most (but not all [42]) applications: we present a small but concrete example of this in Section 2.4.4. Our prototype demonstrates the simplicity of the necessary software support.

We then explore one way for reducing the performance impact of emulated randomization. Our emphasis on the performance aspect of our systems is motivated by our belief that practical, deployable security mechanisms must be efficient enough that the users will not notice their presence, or they will simply not be used. EMUrand, our inline emulator, allows execution of a program in emulated and native modes within the same run. Specifically, we randomize (and emulate) only those portions of the program code that are more likely to be susceptible to remote exploitation. Once the vulnerable sections are identified, it is a simple matter to automatically insert the code for enabling and disabling the emulator. This approach should greatly reduce the performance overhead of emulating the entire OS and application code. EMUrand, by enabling the selective emulation of vulnerable code slices, outperforms our first, straightforward implementation of ISR for $x86$. We explore three different mechanisms to identifying likely vulnerable code slices: heuristics-based, honeypot-directed, and through static source code analysis. These mechanisms can be used in conjunction with each other to identify both a priori and dynamically those portions of the code that are (or may be) exploitable. Other coverage strategies are possible and remain the topic of future research.

Other hardware-based techniques, such as the NoExecute (NX) flag that is available in some recent $x86$-compatible processors [19], have been proposed for dealing specifically with the problem of machine-code injection. In terms of protection, NX and ISR offer very similar type and level of protection, except for the (probably very rare) cases where code and (overwritable) data can be legitimately colocated in the same memory page. However, NX and other hardware-based techniques do not easily translate to other runtime environments and threats (e.g., SQL injection), as ISR does. PaX [29] simulates a per-page NX flag by exploiting the existence of different instruction and data translation lookaside buffers (TLBs) in modern $x86$ processors. This can offer effective protection without hardware support but is nongeneralizable to other environments or even to other processors, including older $x86$ CPUs.

Finally, we apply ISR to the problems of Perl script injection and SQL injection attacks. For the former, we use a modified Perl parser that accepted scripts with randomized keywords, with the randomization key provided as a parameter to the interpreter. For the latter, we create randomized instances of the SQL query language, by randomizing the template query inside the CGI script and the database parser. To allow for easy retrofitting of our solution to existing database systems, we introduce a derandomizing proxy, which converts randomized queries to proper SQL queries for the database. Code injected by the rogue client evaluates to undefined keywords and expressions. When this is the outcome, then standard keywords (e.g., "or") lose their significance, and attacks are frustrated before they can even commence. The performance overhead of our approach is minimal, as we show in Section 4. For interpreted languages, our approach does not lead to any measurable loss in performance relative to non-ISR-enabled execution. Compared to previous techniques, we offer greater transparency to languages, applications, and compilers, as well as a smaller impact on performance.

Paper organization. Sections 2, 3, and 4 describe the application of ISR to binary executables, Perl, and SQL injection, respectively. We discuss some details of the approach, limitations, and future work in Section 5. Section 6 gives an overview of related work aimed at protecting against code-injection attacks.

2 Instruction-Set Randomization for Binaries

Code-injection attacks attempt to deposit executable code (typically machine code, but there are cases where intermediate or interpreted code has been used) within the address space of the victim process and, then, pass control to this code. These attacks can only succeed if the injected code is compatible with the execution environment. For example, injecting $x86$ machine code into a process running on a SUN/SPARC system may crash the process (either by causing the CPU to execute an illegal opcode, or through an illegal memory reference) but will not cause a direct security breach (but will cause a self-inflicted denial of service (DoS), as we discuss in Section 5). Notice that in this example, there may well exist sequences of bytes that will crash on neither processor.

Our approach leverages this observation: we create an execution environment that is unique to the running process, so that the attacker does not know the "language" used and, hence, cannot "talk" to the machine. Conceptually, we can achieve this by applying a reversible transformation between the CPU and main memory. Effectively, we create new instruction sets for each process executing within the same system. Code-injection attacks against this system are unlikely to succeed as the attacker

cannot guess the transformation that has been applied to the currently executing process. Of course, if the attackers had access to the machine and the randomized binaries through other means, they could easily mount a dictionary or known-plaintext attack against the transformation and thus "learn the language." However, we are primarily concerned with attacks against *remote services* (e.g., http, dhcp, DNS, and so on). Vulnerabilities in this type of server allow external attacks (i.e., attacks that do not require a local account on the target system) and, thus, enable large-scale automated exploitation. Protecting against internal users is a much more difficult problem, which we do not address in this work.

In the remainder of this section, we describe ISR at the microlevel (Section 2.1), followed by a description of how an ISR-enabled system operates (Section 2.2). We then provide some details on our implementation (Sections 2.3 and 2.4), including a brief performance evaluation and a discussion on security considerations. We then describe our use of selective ISR (Section 2.5) as a possible way for minimizing the execution overhead of our emulation-based prototype, toward making it usable in certain scenarios (i.e., without requiring hardware support for ISR).

2.1 ISR Operation

The machine instructions for practically all common CPUs consist of *opcodes* that may be followed by one or more arguments. For example, in the Intel $x86$ architecture, the code for the software interrupt instruction is `0xCD`. This is followed by a single 1-byte argument, which specifies the type of interrupt. By changing the relationship between the opcode (`0xCD`) and its operand on the one hand, and the instruction (`INT`) on the other, we can effectively create a new instruction without affecting the processor architecture. Of course, to avoid simple brute force attacks, we need to randomize more than just the opcode; instead, we randomize the whole instruction (i.e., the opcode and all its operands, including immediate values).

For this technique to be effective, the number of possible instruction sets must be relatively large. If the randomization process is driven by a key,[1] we would like this key to be as large as possible. If we consider a generic CPU with fixed 32-bit instructions (like most popular RISC processors), hardware-efficient randomization techniques would consist of XOR'ing each instruction with the key or randomly (based on the key) transposing all the bits within the instruction, respectively. An attacker trying to guess the correct key would have a worst-case work factor of 2^{32} and 32! for XOR and transposition, respectively (notice that $32! >> 2^{32}$). Such a scheme could very efficiently be implemented in the interface between the L2 cache and main memory, as was shown by Rogers et al. [34]. One disadvantage of this simple and efficient encoding is that it is susceptible to the equivalent of known-ciphertext attacks, where the adversary has both the original and the randomized binaries. As we are primarily concerned with remote attacks, in which the adversary does not have a priori access to the system, we believe this is of limited concern. Furthermore, by rerandomizing the binary periodically (e.g., per execution), we can limit the risk of disclosure.

1. The meaning of the term "key" here is similar to its use in modern cryptography, i.e., the security of the randomization process depends on the entropy and secrecy of a random bitstring.

Notice that, in the case of XOR, using a larger block size does not necessarily improve security, since the attacker may be able to attack the key in a piece-meal fashion (i.e., guess the first 32 bits by trying to guess only one instruction, then proceed with guessing the second instruction in a sequence, and so forth). However, we believe that a 32-bit key is sufficient for protecting against code-injection attacks, since the rate at which an attacker can launch these brute-force probing attacks against randomized software is much smaller than in a brute force attack against a cryptographic algorithm. Processors with 64-bit instructions (and thus 64-bit keys, when using XOR for the randomization) are even more resistant to brute force attacks. When using bit transposition within a 32-bit instruction, we need 160 bits to represent the key, although not all possible permutations are valid (the effective key size is $log_2(32!)$). Increasing the block size (i.e., transposing bits between adjacent instructions) can further increase the work factor for an attacker. The drawback of using larger blocks is that the processor must have simultaneous access to the whole block (i.e., multiple instructions) at a time, before it can decode any one of them. Because we believe this greatly increases complexity, we would avoid this scheme on an RISC processor. Unfortunately, the situation is more complicated on the $x86$ architecture, which uses variable-size instructions. The danger is that the effective key size is not really 32 or 64 bits: many of the "interesting" instructions in the $x86$ are 2 bytes long. Thus, an attacker will have to guess two (or four) independent subkeys of 16 bits each. At first glance, it appears that the work factor remains the same ($2^{32} = 2^{16\times2}$), but in fact, it is possible for an attacker to independently attack each of the subkeys, as shown by Sovarel et al. [38]. The implication of this is that on $x86$ the program's text segment must be rerandomized each time a new process is started (we discuss this further in Section 2.4.5).

Finally, note that the security of the scheme depends on the fact that injected code, after it has been transformed by the processor as part of the derandomizing sequence, will raise an exception (e.g., by accessing an illegal address or using an invalid opcode). While this will generally be true, there are a few permutations of injected code that will result in working code that performs the attacker's task. We believe that this number will be statistically insignificant—the same probability as creating a valid buffer-overflow exploit for a known vulnerability by using the output of a random number generator as the injected code.

2.2 System Operation

Let us consider a typical system with an operating system kernel and a number of processes. We seek to defend against code-injection attacks that target applications. Thus, we consider using ISR only when the processor runs in "user" mode. Therefore, the kernel always runs the native instruction set of the processor, which simplifies our task since we do not have to consider the interactions between ISR and the various low-level processor events (e.g., interrupts). Randomization is in effect only while a process is executing in user-level mode.

The code section of each process is loaded from an executable file. The executable file contains the appropriate decoding key in a header, embedded there by the randomizing component of our architecture, described in Section 2.3. For the time being, we will assume that the

executables are statically linked (i.e., there is no code loaded from dynamically linked libraries). We expect only a small number of programs to require static linking, i.e., network services, given our initial focus on remotely exploitable code-injection attacks. We discuss static linking further in Section 5.

The decoding key in the program header is associated with the encoding key used for the encoding of the text segment. Specifically, it would be the same when using XOR as the randomization function, or a key specifying the inverse transposition in the second scheme we discussed earlier. When a new process is loaded from a disk (e.g., as a result of an `exec()` system call), the operating system extracts the key from the header and stores it in the process control block (PCB) structure. In this basic model, executables are randomized once (possibly as the last step in compilation). However, using the same tools we can randomize the executables periodically (using an automated scheduled task), or even at load time, by modifying the loader. These three scenarios represent different trade-offs between minimizing the potential for leakage and exploitation of the randomizing key and performance overhead (primarily process-start latency).

Our approach provides for a special processor register where the decoding key will be stored and a special privileged instruction (called *GAVL*[2]) that allows write-only access to this register. Thus, when the operating system is ready to schedule the execution of a process, the *GAVL* instruction is used to copy the derandomization key from the PCB to the decoding register. To accommodate programs that have not been randomized, we provide a special key that, when loaded via the *GAVL* instruction, disables the decoding process. For programs that have not been randomized, the operating system will load the null decoding key in the PCB. Since the key is always brought in from the PCB, there is no need to save its value during context switches. There is, thus, no instruction to read the value of the decoding register.

When the processor is in kernel mode, the ISR decoding is not active (regardless of the key stored in the *GAV* register). Any transition from user to kernel mode (e.g., hardware or software interrupts) will switch the processor to running the native instruction set. Returning from the interrupt or switching back to user mode reenables the ISR decoding.

As we mentioned earlier, the decoding key is associated with an entire process. It is thus difficult to accommodate dynamically linked libraries, as these would either have to be encoded as they are loaded from disk or be encoded and copied into a completely disjoint set of memory pages in the case of already memory-resident libraries. In both cases, the memory occupied by the encrypted code for the libraries will not be shareable with other processes, or all the processes would have to share the same key (the one used by the libraries). Since neither approach is appealing, we decided to require statically linked executables. In practice, we would seek to randomize (and thus statically link) only those programs that are directly exposed to remote exploits, i.e., network daemons, thus reducing the overall performance and management impact of static linking to the system.

2. GAVL: Gaurav, Angelos, Vassilis, plus L because it is a load instruction.

For example, a system may be running a Web server, while also supporting the website developers. The Web server application and possibly the ssh daemon would be randomized, since both programs are interacting with the outside world. Other processes such as editors, spell checkers, and so forth, may not need to be randomized and, hence, can be dynamically linked.

2.3 Randomized ELF Executables

The Executable and Linking Format (ELF) is the standard file format used with the *gcc* compiler and associated utilities like the assembler and linker in many architectures, for encoding executable and library files. ELF completely separates code and data sections, including read-only control data such as jump tables. This was very useful for us to be able to single out the executable sections in an ELF executable so that we could then carry out their block randomization.[3] We modified several utilities from the GNU *binutils* package to implement our randomization process. In particular, the *objcopy* utility handles processing of the ELF headers, but it also provided a reference to a byte array (representing the machine instruction block) for each given code section in the file. We were then able to take advantage of this fact by randomizing each 16-bit block in this array before letting the rest of the original program continue producing the target file.

2.4 $x86$ ISR Implementation

To determine the feasibility of our approach, we built a prototype of the proposed architecture using the *bochs* emulator [8] for the $x86$ processor family. As we discussed in Section 2.1, randomization on the $x86$ is more complicated than with RISC-type processors because of its use of variable-size instructions. However, we decided to implement the randomization for the $x86$ both to test its feasibility in a worst-case scenario and because of the processor's wide use.

2.4.1 Runtime Environment

Even without a specially modified CPU, the benefits of randomized executables can be reaped by combining a sandboxed environment that emulates a conventional CPU with the instruction randomization primitives discussed earlier. Such a sandboxing environment would need to include a CPU emulator like *bochs* [8], its own operating system, and the process(es) we wish to protect.

2.4.2 Bochs Modifications

Bochs is an open-source emulator of the $x86$ architecture. Since it interprets each machine instruction in software, *bochs* allows us to perform any restoration operations on the instruction bytes as they are fetched from instruction memory. The core of *bochs* is implemented in the function *cpu_loop()* that uses another function, *fetchDecode()*, passing a reference into an array representing a block of instruction code. The *fetchDecode()* function incrementally extracts a byte from that array until it can complete decoding of the current instruction. This behavior closely simulates the $i486$ and *Pentium* processors, with their instruction "prefetch streaming buffers." On the $i486$, this buffer held the next

3. Note that this data-text separation may not hold for other binary formats or compilers. In those cases, we need to avoid "randomizing" embedded read-only data during the randomization pass.

16 bytes worth of instructions; on later processors, this has typically been 32 bytes. We carry out our derandomizing of this instruction at the beginning of *fetchDecode()*, based on the decoding key value that is currently loaded on the special processor register, as discussed in Section 2.

2.4.3 Single-System Image Prototype

To simplify the creation and evaluation of our prototype, we adopted the techniques we used to construct embedded systems for VPN gateways [33]. We use automated scripts to produce compact (2- to 4-Mbyte) bootable single-system images that contain a system kernel and applications. We achieve this by linking the code of all the executables that we wish to be available at runtime in a single executable using the *crunchgen* utility. The single executable alters its behavior depending on the name under which it is run (*argv[0]*). By associating this executable with the names of the individual utilities (via file system hard links), we can create a fully functional */bin* directory where all the system commands are accessible as apparently distinct files. This aggregation of the system executables in a single image greatly simplifies the randomization process, as we do not need to support multiple executables or dynamic libraries. Although this also greatly constrains the real-world applicability of our technique, we felt this was an acceptable compromise for a prototype implementation. The root of the runtime file system, together with the executable and associated links, is placed in a RAM disk that is stored within the kernel binary. The kernel is then compressed (using *gzip*) and placed on a bootable medium (in our case, a file that *bochs* considers to be its boot device). This file system image also contains the */etc* directory of the running system in uncompressed form, to allow easy configuration of the runtime parameters.

At boot time, the kernel is copied from the boot image to *bochs'* main memory and is uncompressed and executed. The file system root is then located in the RAM disk. The */etc* directory is copied to the RAM disk from the temporarily mounted boot partition. The system is running entirely off the RAM disk and proceeds with the regular initialization process. This organization allows multiple applications to be combined within a single kernel where they are compressed, while leaving the configuration files in the */etc* directory on the boot partition. Thus, these files can be easily accessed and modified. This allows a single image to be produced and the configuration of each sandbox to be applied to it just before it is copied to this separate boot partition.

2.4.4 Performance

Since our goal was simply to demonstrate the feasibility of our approach, we chose a few, very simple benchmarks. Generally, interpreting emulators (as opposed to virtual machine emulators, such as VMWare) imposes a considerable performance penalty; depending on the application, the slowdown can range from one to several orders of magnitude. This makes such an emulator generally inappropriate for high-performance applications, although it may be suitable for certain high-availability environments.

Table 1 compares the time taken by the respective server applications to handle some fairly involved client activity. The times recorded for the *ftp* server were for a client carrying out a sequence of common file and directory operations, viz., repeated upload and download of an $\approx$ 200-Kbyte file, and creation, deletion, and renaming of directories, and generating directory listings by means of an automated script. This script was executed 10 times, and the execution times were averaged to produce the times listed. This *ftp* result illustrates that a network I/O-intensive process does not suffer execution time slowdown proportional to the reduction in processor speed. The *sendmail* numbers, taken from the mail server's logging file, represent the overall time taken to receive 100 short e-mails ($\approx$ 1 Kbyte each) from a remote host.

TABLE 1
Experimental Results: Average Execution Times (in Seconds) for Identical Binaries on Bochs and a Regular Linux Machine (the Same One that Hosted the Emulator)

	ftp	**sendmail**	**fibonacci**
bochs	39.0s	$\approx 28s$	$\approx 93s$
linux	29.2s	$\approx 1.35s$	0.322s

The performance numbers of individual runs were within 10 percent of the listed averages.

The last column demonstrates the significant slowdown incurred in the emulator when running a CPU-intensive application (as opposed to the I/O-bound jobs represented in the first two examples), such as computation of the *fibonacci* numbers. However, this only helps confirm the existence of real-world applications for emulators. All the applications were compiled with the *-static-falign-labels* option for *gcc*, with zero optimization.

Emulator-based approaches have also been proposed in the context of intrusion and anomaly detection [17], as well as one way to retain backward compatibility with older processors—often exhibiting better performance. However, to make our proposal fully practical, we will need to modify an actual CPU.

2.4.5 Security Considerations

Performing code injection in a few vulnerable applications that used ISR caused the targets to terminate with a segmentation violation or illegal opcode exception. Barrantes et al. did a comprehensive study on the types of faults that ISR would cause a compromised process to exhibit, showing that such processes execute at most five $x86$ instructions before causing an exception [5]. These instructions are effectively random bytes derived from the attacker's injected code through the (unpredictable to the attacker) derandomization process.

Brute force attacks. One obvious way to attack our system is through brute force code injection: the adversary simply tries all possible randomization keys on the attack payload they wish to inject. For that reason, it is important to maximize the key space of our technique. We believe 32 bits to be sufficient for our purposes, since attackers can try these guesses at a relatively slow rate (compared to brute-forcing a cryptographic key) because they need to do so remotely. The fact that the targeted process will stop for at least a short period of time further limits the probing rate. If the randomization key is changed every time the process is restarted (as we discuss next), then the attack becomes a pure game of chance where the attacker has 2^{-32} probability of success in any given trial. Furthermore, as Cox et al. showed [16], one can construct quorum architectures where

each replica is randomized differently (with a different key) to defeat all guessing attacks.

When to randomize. As Sovarel et al. [38] demonstrated, it is possible to extract information about the randomization key such that only a small number of attempts is needed to succeed in injecting code. Similar attacks were shown against address-space randomization [35]. If ISR is used in a production environment, the randomization process should take place when the program is executed (load time). Under such a scheme, the code segment of the program is randomized as it is copied into the swap area. While this increases the load time latency, the overhead is low (adding approximately less than 10 percent to the startup time of a process). For long-lived processes (such as most network-facing daemons), this cost is amortized over a long period of time.

In the case of server processes that *fork()* a copy of the parent process to handle each request, the attacker can attempt multiple guesses against the same key by injecting code consisting of short instructions (assuming a variable instruction-length architecture, like the $x86$). In such cases, the *fork()* system call itself can be modified so that if the process is employing ISR, the text segment is actually copied (rather than copy the page table entries of the parent) and rerandomized. It is worth noting that such failures can be used to perform near-real-time forensic analysis to identify the vulnerability the attacker is exploiting and to generate a signature [14], [45], [26], [23].

Known ciphertext attacks. Since program code is highly structured, an attacker with access to the randomized code segment of a running process can easily determine the randomization key and thus create valid attack payloads. Such access may be easy (depending on system configuration) for an attacker that already has local access to the system, e.g., through the */proc* interface, and will allow a privilege escalation. One way to mitigate this problem is to use a strong encryption algorithm as the randomization primitive, possibly incurring a performance hit. We believe our technique to be primarily applicable in deterring remote attackers seeking to use code injection.

Even in that scenario, however, it is imperative that the application does not expose parts of its code segment, e.g., by improperly returning such information as part of an error message. Although we believe this not to be a significant problem for binary applications, it is an important issue in other domains where ISR may be used, such as SQL injection prevention. As we discuss in Section 4, one way of dealing with this problem is through a scrubbing proxy (an approach that lends itself to SQL injection but is probably not generalizable).

2.5 EMUrand: Selective Instruction Set Randomization

Our solution to the performance problem of software-based ISR is to provide a lightweight, minimally invasive emulator that can switch freely between derandomizing the instruction stream and normal execution of the instruction stream on the underlying hardware. We may then employ several selection strategies to identify code sections that are likely to contain a vulnerability and wrap this code with tags that invoke the emulator.

We take three approaches to address the selection problem. First, we take advantage of the CoSAK study,[4] which used statistical analysis to determine the probability that a particular function was vulnerable to code-injection attacks. This probability is based on the function's location in a call graph relative to input functions (like the *read()* system call). The study analyzed open-source applications with known vulnerabilities. The researchers constructed a call graph of each application and then counted back from input routines to functions where a vulnerable buffer was declared. On average, the distance in the call graph was 5.5 function calls. The average number of functions that fall within this range is approximately 3 percent.

Second, we can utilize an approach suggested by the Worm Vaccine project [36] to set up an appropriately instrumented "victim" application to act as a honeypot. Once the victim receives an exploit string that causes a buffer overrun or similar code-injection attack, the instrumentation (a modified version of ProPolice[5]) precisely identifies the vulnerable buffer and function. This information can then be used to automatically protect (via EMUrand) that section of code in the absence of a patch.

Third, we employ static source code analysis tools like RATS[6] or Splint [18] to identify sections of code that are likely to contain some well-known vulnerability. Static analysis is far from perfect: it produces a number of false positives and may miss some vulnerabilities (false negatives). However, such tools help us target the use of EMUrand to likely vulnerable code.

These selection strategies are useful heuristics. As such, they are prone to a certain amount of error. Finding and fixing bugs through regular code audits should proceed regardless of the protection mechanism in use. Even though the selection strategies can identify potential vulnerabilities, the application will still require some protection during the window in which a developer fixes these new vulnerabilities and deploys the fix to the application. Even if the attempt to fix every identified vulnerability is made, there is no guarantee that the solution will be free of defects. In contrast, ISR will function regardless of the particular content of an attack or method of injection. In cases where the identification of a vulnerability is fairly specific, such as in the approach proposed by Sidiroglou and Keromytis [36], ISR can serve as a first line of defense while a patch is being generated by a developer or applied by a system administrator.

2.5.1 EMUrand Implementation

With the absence of a programmable processor, unscrambling randomized instructions requires software emulation. Since current $x86$ emulators do not perform either ISR or selective emulation, we designed EMUrand to bridge the gap. EMUrand runs inside of an executable and only needs to maintain enough state to process instructions: the general, segment, eflags, and FPU registers. Memory management and other OS-specific tasks operate normally.

To enable EMUrand, we provide a statically linked library that defines special tags (combinations of macros and function calls), which mark the beginning and end of selective emulation. Upon entering the vulnerable section of code, the emulator captures the program state (as defined

4. http://serg.cs.drexel.edu/cosak/index.shtml.
5. http://www.trl.ibm.com/projects/security/ssp.
6. http://www.securesw.com/download_rats.htm.

```
void foo() {
    int a = 1;
    emulate_begin(emurand_args);
    a++;
    emulate_end();
    printf("a = %d\n", a);
}
```

Fig. 1. EMUrand invocation and termination tags.

above) and processes all instructions inside the area designated for ISR. When the program counter references the first instruction below the bounds of emulation, the virtual processor copies its internal state back to the program. While registers are explicitly updated, memory updates have implicitly been applied throughout the execution of EMUrand. The program, unaware of the instructions executed by the virtual processor, is equipped to continue normal processing.

We implemented EMUrand to validate the practicality of selective emulation. The majority of development time was dedicated to the functions that simulated $x86$ instructions for 32-bit and 16-bit operand sizes. Integrating EMUrand into an existing application is fairly straightforward. As Fig. 1 shows, special tags are wrapped around the segment of code to be emulated.

The C macro *emulate_begin()* moves the program state (general, segment, eflags, and FPU registers) into an emulator-accessible global data structure to capture state immediately before EMUrand takes control. EMUrand uses this data to initialize the virtual registers. When this setup completes, *emulate_begin()* obtains the memory location of the first instruction following the call to itself. The instruction address is the same as the return address and can be found in the activation record of *emulate_begin()*, four bytes above its base stack pointer. The *emulate_begin()* function takes as argument the key used for derandomization.

EMUrand operation. The main loop of the emulator fetches, decodes, executes, and retires one instruction at a time. Before fetching an instruction, derandomization takes place, in the same manner as we discussed in Section 2. The fetch/decode/execute/retire cycle of instructions continues until either *emulate_end()* is reached, or when the emulator detects that control is returning to the parent function. If the emulator does not encounter unrandomized opcodes, upon completion the emulator's instruction pointer reaches *emulate_end()*. To enable the program to continue execution at this address, the return address of the *emulate_begin()* activation record is replaced with the current value of the instruction pointer. By executing *emulate_end()*, the emulator's environment is copied to the program registers and execution continues under normal conditions. If an exception arises during emulation, it is assumed to be due to injected (unrandomized) code; EMUrand then prints diagnostics and terminates program execution.

The emulator is designed to execute in user mode, so system calls cannot be computed directly without kernel-level permissions. Therefore, when the emulator decodes the Linux system call interrupt or the *sysenter* instruction, it must release control to the kernel. However, before the kernel can successfully execute the system call, the program state needs to reflect the virtual registers arrived at by

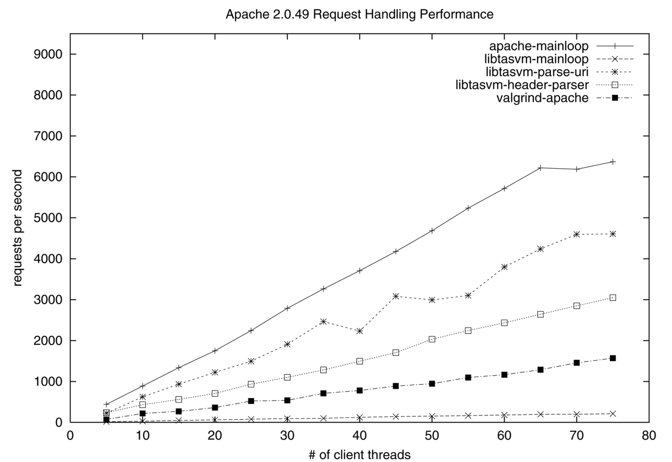

Fig. 2. *Timing of main request processing loop including EMUrand and Valgrind.* Valgrind appears to run better than EMUrand when executing the entire request loop. However, selective emulation still performs better than Valgrind and request processing is sustainable. EMUrand is denoted by the "libtasvm" lines (the name of the library containing EMUrand).

EMUrand. The emulator saves the real registers and replaces them with its own values. EMUrand issues the interrupt, and the kernel processes the system call. Once control returns, the emulator updates its registers and restores the original values in the program's registers.

2.5.2 EMUrand Performance Evaluation

While the main goal of previous work on ISR was to simply demonstrate the efficacy of the approach, a critical component of our research and experimentation has been to identify and construct an environment that strikes the right balance between emulation overhead and protection against code-injection attacks. Keeping in mind the selection strategies we discussed earlier, we evaluated the performance impact of EMUrand by instrumenting the Apache 2.0.49 Web server and performing microbenchmarks on utilities such as *ls*, *cat*, and *cp*.

The machine hosting Apache was a Pentium III at 1 GHz with 512 Mbytes of memory running RedHat Linux with kernel 2.4.20. The client machine was a dual Pentium II at 350 MHz with 256 Mbytes of memory running RedHat Linux 8.0 with kernel 2.4.18smp. Both emulated and nonemulated versions of Apache were compiled with the *-enable-static-support* configuration option. EMUrand was compiled with the "*-g-static-fno-defer-pop*" flags.

We used the Apache *flood* testing tool to evaluate how quickly both the nonemulated and emulated versions of Apache would process requests. In our experiments, performance was measured by the total number of requests processed (see Fig. 2). The value for total number of requests per second is extrapolated (by *flood*'s reporting tool) from a smaller number of requests sent and processed within a smaller timeslice; the value should not be interpreted to mean that the test Apache instances and test hardware actually served some 6,000 requests per second.

Emulation of Apache inside Valgrind. To get a sense of the performance degradation imposed by running the entire process inside an emulator other than EMUrand (and thus determine the inefficiency, if any, inherent to our implementation of emulation), we tested Apache running in

TABLE 2
Occurrences of RATS Alerts in Apache

Alert Type	Alerts	Files	Functions
fixed size buffer	58	16	26
string format	12	3	5
strcpy, getenv	18	2	4

TABLE 3
Emulated ISR Performance for Various Command-Line Utilities

Test Type	trials	mean (s)	Std. Dev.	Min	Max	Instr. Emulated
ls (non-emu)	25	0.12	0.009	0.121	0.167	0
ls (emu)	25	42.32	0.182	42.19	43.012	18,000,000
cp (non-emu)	25	16.63	0.707	15.80	17.61	0
cp (emu)	25	21.45	0.871	20.31	23.42	2,100,000
cat (non-emu)	25	7.56	0.05	7.48	7.65	0
cat (emu)	25	8.75	0.08	8.64	8.99	947,892

Valgrind version 2.0.0 on the Linux test machine that hosted Apache for our EMUrand test trials. Valgrind, which was used in RISE [4], has two notable features that improve performance over our full emulation of the main request loop. First, Valgrind maintains a cache of translated instructions while EMUrand translates each encountered instruction from scratch. Second, Valgrind performs some optimizations to avoid redundant load, store, and register-to-register move operations. We ran Apache under Valgrind with the tool *Memcheck* and the $-trace-children = yes$ option. While Valgrind performed better than our emulation of the full request processing loop, it did not perform as well as our emulated slices, as shown in Fig. 2.

Optimizations to EMUrand to maintain a cache of already-translated instructions should boost performance significantly. We did not include such a cache in order to minimize the memory footprint of EMUrand, although it may be advantageous to do so in the future. For example, the Valgrindized version of Apache is 10 times the size of the regular Apache image, while Apache running with EMUrand is not detectably larger.

Full emulation and baseline performance. We first demonstrate that emulating the bulk of an application entails a significant performance impact. In particular, we emulated the main request processing loop for Apache and compared our results against a nonemulated Apache instance. In handling the processing of this main loop, the emulator executed roughly 213,000 instructions. In order to get a more complete sense of this performance impact, we timed the execution of the request handling procedure for both the nonemulated and fully emulated versions of Apache by embedding calls to *gettimeofday()* where the emulation functions were (or would be) invoked.

For our test machines and sample loads, the nonemulated Apache spent some 6,300 μs to perform the work in the main loop function. The fully instrumented loop running in the emulator spends an average of 277,927 μs (or 277 ms) per request in that particular code section. For comparison, we also timed Valgrind's execution of this section of code; after a large initial cost (presumably to perform the initial translation and fill the internal instruction cache), Valgrind executes the section with a 34,193 μs average.

Selective emulation. We used the RATS tool to identify possible vulnerable sections of code in Apache 2.0.49. The tool identified roughly 270 lines of code, the majority of which contained fixed-size local buffers. To gain some intuition about how much of the application actually contained some of these alerts, we ran RATS on the server code module and produced the results in Table 2.

The main request handling logic in Apache 2.0.49 begins in the *ap_process_http_connection()* function. The effective work of this function is carried out by two subroutines: *ap_read_request()* and *ap_process_request()*. The *ap_process_request()* function is where Apache spends most of its time during the handling of a particular request. In contrast, the *ap_read_request()* function accounts for a smaller fraction of the request handling work. We decided to emulate subroutines of each function in order to assess the impact that selective emulation would have.

We constructed a partial call tree and chose the *ap_parse_uri()* function (invoked via *read_request_line()* in *ap_read_request()*) and the *ap_run_header_parser()* function (invoked via *ap_process_request_internal()* in *ap_process_request()*). The emulator processed approximately 358 and 3,229 instructions, respectively, for these two functions. In each case, the performance impact, as expected, was much less than the overhead incurred by needlessly emulating the entire work of the request processing loop.

Microbenchmarks. Using the client machine from the Apache performance tests, we ran some microbenchmarks to gain a broader view of EMUrand's performance impact. We selected some shell utilities and measured their performance for large workloads running both with and without EMUrand. We issued an "*ls-R*" command on the root of the Apache source code with both stderr and stdout redirected to */dev/null* in order to reduce the effects of screen I/O. We then used *cat* and *cp* on a large file (also with screen I/O redirected to */dev/null*) (Table 3).

As expected, there is a large impact on performance when emulating the majority of an application. Our experiments demonstrate that only emulating potentially vulnerable sections of code offers a significant advantage over emulation of the entire system. While our prototype $x86$ emulator, EMUrand, is a fairly straightforward implementation, it can gain further performance benefits by using Valgrind's technique of caching already translated instructions. With some further optimizations, EMUrand is a viable and practical approach for protecting code with ISR. In fact, Bruening et al. [9] outline several ways to optimize emulators; their approaches reduce the performance overhead (as measured by two SPEC2000 benchmarks, crafty and vpr) from a factor of 300 to about 1.7. Their optimizations include caching basic blocks, linking direct and indirect branches, and building traces.

3 RANDOMIZED PERL

The attractiveness of the ISR concept lies in its generality and applicability in different environments and runtimes. To demonstrate its versatility, we implemented ISR for Perl.

In the Perl prototype, we randomized all the keywords, operators, and function calls in a script. We did so by

appending a random nine-digit number ("tag") to each of these. For example

```
foreach $k (sort keys %$tre) {
  $v = $tre - > {$k};
  die "duplicate key $k\ n"
     if defined $list{$k};
  push @list, @{ $list{$k} };
}
```

by using "123456789" as the tag, becomes

```
foreach123456789 $k (sort123456789 keys %$tre)
{
  $v =1234567889 $tre->{$k};
  die123456789 "duplicate key $k\ n"
    if123456789 defined123456789 $list{$k};
  push123456789 @list, @{ $list{$k} };
}
```

Perl code injected by an attacker will fail to execute, since the parser will fail to recognize the (missing or wrong) tag.

We implemented the randomization by modifying the Perl interpreter's lexical analyzer to recognize keywords followed by the correct tag. The key is provided to the Perl interpreter via a command-line argument, thus allowing us to embed it inside the randomized script itself, e.g., by using "#!/usr/bin/perl -r123456789" as the first line of the script. Upon reading the tag, the interpreter zeroes it out so that it is not available to the script itself via the ARGV array. These modifications were fairly straightforward and took less than a day to implement. To generate the randomized code, we used the Perltidy [30] script, which was originally used to indent and reformat Perl scripts to make them easier to read. This allowed us to easily parse valid Perl scripts and emit the randomized tags as needed.

One problem we encountered was the use of external modules. These play the role of code libraries and are usually shared by many different scripts and users. To allow their sharing in randomized scripts, we use two tags: the first is supplied by the user via the command line, as discussed above, while the second is a system-wide key known to the Perl interpreter. We extended the lexical analyzer to accept either of these tags. Using this scheme, the administrator can periodically randomize the system modules, without requiring any action from the users. Also, note that we do not randomize the function definitions themselves. This allows scripts that are not run in randomized mode to use the same modules. Although the size of the scripts increases considerably due to the randomization process, some preliminary measurements indicate that performance is unaffected.

A similar approach can counter attacks against Web CGI scripts that dynamically generate SQL queries to a back-end database. Such attacks can have serious security and privacy impact [11]. In that case, we would modify the SQL interpreter similar to what we described for Perl and generate randomized SQL queries in the CGI script. To avoid modifying the database front end, we can use a validating proxy that intercepts randomized SQL queries. If the queries are syntactically correct (i.e., appropriately randomized), they are derandomized and passed on to the database.

We can also use randomization with CGI scripts that issue Unix shell commands. By randomizing the shell interpreter, we can avoid injection attacks [10]. In this scenario, we would also randomize the program names, e.g.,

```
#!/bin/sh

if987654 [ x$1 = =987654 x""; then987654
  echo987654 "Must provide directory name."
  exit987654 1
fi987654

/bin/ls987654 -l $1
exit987654 0
```

In all cases, we must hide low-level (e.g., parsing) errors from the remote user, as these could reveal the tag and thus compromise the security of the scheme. Other applications of ISR include VBS and other email- or Web-accessible scripting languages.

4 Protection against SQL Injection Attacks

We applied ISR to the problem of defending database-driven Web applications from SQL injection attacks, in part to further demonstrate its versatility in protecting against injection attacks in a diverse set of execution environments.

Injecting SQL code into a Web application requires little effort by those who understand both the semantics of the SQL language and CGI scripts. Many Web applications combine user input with predefined queries and then transmit the result to the database for execution. Unless developers properly design application code to protect against unexpected data input, alteration of the database structure, corruption of data, or revelation of private and confidential information may inadvertently occur.

For example, consider a login page of a CGI application that expects a username and the corresponding password. When the credentials are submitted, they are inserted within a query template such as the following:

```
"select * from mysql.user
   where username =' " . $uid . " ' and
      password = password (' ". $pwd . " ');"
```

Instead of a valid username, the malicious user sets the $uid variable to the string

```
' or 1 = 1; - -'
```

causing the CGI script to issue the following SQL query to the database:

```
"select * from mysql.user
  where username = '' or 1 = 1; - -'' and
     password = password ('_any_text_');"
```

The single quotes balance the quotes in the predefined query, and the double hyphen comments out the remainder of the SQL query. Therefore, the password value is irrelevant and may be set to any character string. The result set of the query contains at least one record, since the "where" clause evaluates to true. If the application

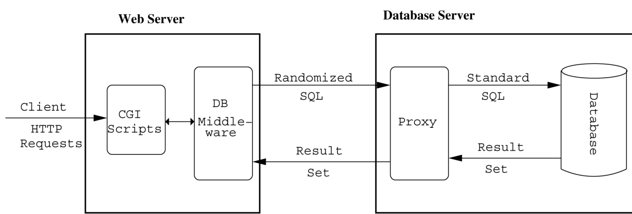

Fig. 3. SQLrand system architecture.

identifies a valid user by testing whether the result set is nonempty, the attacker can bypass the security check.

Our solution extends the application of ISR to the SQL language: the SQL standard operators (including keywords, mathematical operators, and other invariant language tokens) are manipulated by appending a random integer to them, one that an attacker cannot easily guess. Any malicious user attempting an SQL injection attack would be thwarted because the user input inserted into the "randomized" query would always be classified as a set of nonoperators, resulting in an invalid expression.

Essentially, we have introduced a new set of keywords to SQL that will not be recognized by the database's SQL interpreter. A difficult approach would be to modify the database's interpreter to accept the new set of keywords. Attempting to change its behavior, however, would be a daunting task. Furthermore, a modified database would require all applications submitting SQL queries to conform to its new language. Although dedicating the database server for selected applications might be possible, the random key would not be varied among the SQL applications using it. Ideally, having the ability to vary the random SQL key, while maintaining one database system, grants a greater level of security by making it difficult to subvert multiple applications by successfully attacking a single instance.

Our design consists of a proxy that sits between the client and database server (see Fig. 3). Note, however, that the proxy may be on a separate machine. By moving the derandomization process outside the Database Management System (DBMS) to the proxy, we gain flexibility, simplicity, and security. Multiple proxies using unique random keys to decode SQL commands can be listening for connections on behalf of the same database, while allowing disparate SQL applications to communicate in their own "tongue." The interpreter is no longer bound to the internals of the DBMS. The proxy's primary obligation is to decipher the random SQL query and then forward the SQL command with the standard set of keywords to the database for computation. Another benefit of the proxy is the concealment of database errors, which may reveal the random SQL keyword extension to the user. A typical attack consists of a simple injection of SQL, hoping that the error message will disclose a subset of the query or table information, which may be used to deduce hidden database properties. By stripping the randomization tags in the proxy, we need not worry about the DBMS inadvertently exposing such information through error messages; the DBMS itself never sees the randomization tags. Thus, to ensure the security of the scheme, we only need to ensure that no messages generated by the proxy itself are ever sent to the DBMS or the front-end server. Given that the proxy itself is fairly simple, it seems possible to secure it against attacks. If the proxy is compromised, the database remains safe, assuming that other security measures are in place.

To assist the developer in randomizing his SQL statements, we provide a tool that reads an SQL statement(s) and rewrites all keywords with the random key appended. For example, an SQL query, which takes user input, may look like the following:

```
select gender, avg(age)
    from cs101.students
      where dept = %d
    group by gender
```

The utility will identify the six keywords in the sample query and append the key to each one (e.g., when the key is "123"):

```
select123 gender, avg123 (age)
    from123 cs101.students
      where123 dept = %d
    group123 by123 gender
```

This SQL template query can be inserted into the developer's Web application. The proxy, upon receiving the randomized SQL, translates and validates it before forwarding it to the database. Note that the proxy performs simple syntactic validation—it is otherwise unaware of the semantics of the query itself.

4.1 SQLrand Implementation

We built a proof-of-concept proxy server that sits between the client script and SQL server, derandomizes requests received from the client, and conveys the query to the server. If an SQL injection attack occurs, the proxy's parser will fail to recognize the randomized query and will reject it. The two primary components were the derandomization element and the communication protocol between the client and database system. In order to derandomize the SQL query, the proxy required a modified SQL parser that expected the suffix of integers applied to all keywords. As a "middle man," it had to conceal its identity by masquerading as the database to the client and vice versa. Although our primary implementation focused on CGI scripts as the query generators, a similar approach applies when using the Java database access framework (JDBC), as we describe in Section 4.2.

The randomized SQL parser utilized two popular tools for writing compilers and parsers: flex and yacc. Capturing the encoded tokens required regular expressions that matched each SQL keyword (case-insensitive) followed by zero or more digits. (Technically, it did not require a key; practically, it needs one.) If properly encoded, the lexical analyzer strips the token's extension and returns it to the grammar for query reassembly. Otherwise, the token remains unaltered and is labeled as an identifier. By default, flex reads a source file, but our design required an array of characters as input. To override this behavior, the YY_INPUT macro was redefined to retrieve tokens from a character string introduced by the proxy. During the parsing phase, any syntax error signals the improper construction of an SQL query using the preselected random key. Either the developer's SQL template is incorrect or the user's input includes unexpected data, whether good or bad. On encountering this, the parser returns NULL;

otherwise, in the case of a successful parse, the derandomized SQL string is returned. The parser was designed as a C library.

We used MySQL, a popular open-source database system, to create a sample customer database. The record size of the tables ranged from 20 to a little more than 11,000 records. These sample tables were used in the evaluation of benchmark measurements described in Section 4.3. After completing the parser and creating this database, we defined a communication protocol between the proxy and the MySQL database.

Since the proxy will act as a client to the database, the C API library was suitable. One problem existed: the mysqlclient C library does not have a server-side counterpart for accepting and disassembling the MySQL packets sent using the client API. Therefore, we needed to analyze the MySQL protocol and incorporate it into the proxy. Unfortunately, we did not find official documentation; however, a rough sketch of the protocol existed, which satisfied the requirements of the three primary packets: the query, the error, and the disconnect packets.

The query packet carries the actual request. The quit message is necessary in cases where the client abruptly disconnects from the proxy or sends the proxy an invalid query. In either case, the proxy becomes responsible for disconnecting from the database by issuing the quit command on behalf of the client. Finally, the error packet is only sent to the client when an improper query generates a syntax error, indicating a possible injection attack.

The client application needs only to define its server connection to redirect its packets through the proxy rather than directly to the database. In its connection method, this is achieved simply by changing the port number of the database to the port where the proxy is listening. After receiving a connection, the proxy in turn establishes a connection with the database and transmits messages it receives from the client. If the command byte of the MySQL packet from the client indicates the packet contains a query, the proxy extracts the SQL and passes it to the interpreter for decoding. When unsuccessful, the proxy sends an error packet with a generic "syntax error" message to the client and disconnects from the database. On the other hand, a successful parsing of the SQL query produces a translation to the derandomized syntax. The proxy overwrites the original, randomized query with the standard query that the database is expecting into the body of the MySQL packet. The packet size is updated in the header and pushed out to the database. The normal flow of packets continues until the client requests another query.

The API libraries define some methods that will not work with the proxy, as they hardcode the SQL query submitted to the database. For example, mysql_list_dbs() sends the query "SHOW databases LIKE <wild-card-input>." Without modification to the client library, the workaround would be to construct the query string with the proper randomized key and issue the mysql_query() method. Presently, binary SQL cannot be passed to the proxy for processing; therefore, mysql_real_query() must be avoided.

4.2 JDBC-Based SQL Randomization

Our JDBC-based system consists of an offline tool used to randomize the SQL statements inserted into a Java application (e.g., a Java servlet) and an online module that is part of the JDBC driver. The second component both parses and derandomizes SQL statements and queries. Web servers delegate HTTP requests involving dynamic content to an application server that utilizes our modified JDBC driver to obtain data from the database. The database answers the query, and the application server passes an HTTP response back to the client via the Web server. In our system, the SQL statements in the application server have been randomized and are derandomized by the JDBC driver before being sent to the database. Any SQL injected by the attacker will fail to derandomize correctly, causing the driver to raise an exception; the injected SQL never gets to the database.

4.2.1 Implementation

Our proof-of-concept system utilizes a simple but incomplete parsing strategy for SQL statements. Our implementation contains a utility to randomize SQL statement strings (from either standard input or a file) and modifications to the JDBC driver to configure the driver with the randomization key and derandomize the text of SQL statements passed to it via its API. Finally, our system includes a sample test application that uses the modified driver and accepts SQL statements from an interactive command prompt. This test utility can be used to interactively attempt SQL injection and observe how the driver refuses to accept the modified statements.

We constructed a Java class to accept text SQL statements as input and randomize them with a given key. Web application developers would use this tool to randomize their SQL queries before inserting them into the application's code. We modified the JDBC driver to accept a new configuration parameter that contains the randomization key. When the driver is asked to execute a statement or query, the driver checks if this configuration parameter exists. If it does, the driver parses the SQL statement and attempts to strip off the key from each SQL keyword. If this process fails, the driver raises an exception for the calling code.

4.2.2 Limitations

Dealing with stored procedures is a difficult issue, as these SQL statements are stored directly in and invoked by the database itself. It is impossible to derandomize them without changing the SQL parsing logic in the database. In addition, changing the randomization key is a potential issue. One solution is to store queries in an external data source (e.g., an XML file) that the application reads during execution. This content can be randomized during runtime under a different key.

SQL randomization is meant to work in an environment where the attacker does not have direct control over the derandomization process. If an attacker controls this component (in our case, the JDBC driver), they can easily discover the key.

Finally, our system does not intercept errors generated by a failure to derandomize SQL statements; rather, its purpose is to detect an attack and then notify the Web application by generating an `SQLException`. Instances of `SQLException` occur normally during JDBC operations, and applications generally expect to handle them. Web

applications should be wary about displaying the error messages. Doing so may reveal the randomization key.

4.3 SQLrand Evaluation

We evaluated the practicality of using a proxy to derandomize encoded SQL by considering the efficacy of the procedure (by observing if it prevents known SQL injection vulnerabilities) as well as the overhead introduced by the proxy.

4.3.1 Qualitative Evaluation

First, we wrote a sample CGI application that allowed a user to inject SQL into a "where" clause that expected an account ID. With no input validation, one can easily inject SQL to retrieve account information concerning all accounts. When using the SQLrand proxy, the injected statement is identified and an error message issued, rather than proceeding with the processing of the corrupted SQL query.

After testing the reliability of the proxy in this simple example, the next step was to identify an SQL injection vulnerability in an existing application. A Web bulletin board, phpBB v2.0.5, presented an opportunity to inject SQL into viewtopic.php, revealing the password of a user 1 byte at a time. After the attack was replicated in the test environment, the section of vulnerable SQL was randomized and the connection was redirected through the proxy. As expected, the proxy recognized the injection as an invalid SQL code and did not send it to the database. The phpBB application did not succumb to the SQL injection attack as verified without the proxy. However, it was observed that the application displays an SQL query to the user by default when zero records are returned. Since an exception does not return any rows, the proxy's encoding key was revealed. Again, ISR still requires good coding practices. If a developer chooses to reveal the SQL under certain cases, there is little benefit to the randomization process.

Another content management application prone to SQL injection attacks, PHP-Nuke, depends on the magic_quotes_gpc option being enabled. Without this setting, several modules are open to attack. Even with the option set, injections on numeric fields are not prevented because the application does not check for numeric input. For example, when attempting to download content from the PHP-Nuke application, the download option d_op is set to "getit" and accepts an unchecked, numeric parameter name "lid." It looks up the URL for the content from the download table based on the lid value and sets it in the HTTP location header statement. If an attacker finds an invalid lid (determined by PHP-Nuke reloading its home page) and appends "union select pass from users_table" to it, the browser responds with an error message stating that the URL had failed to load, thus revealing the sensitive information. However, when we used the proxy, injection attacks in the affected module were averted. These vulnerabilities exist in other PHP-Nuke modules and can be averted by using the proxy.

4.3.2 Performance Evaluation

Next, we quantified the overhead imposed by SQLrand. We designed an experiment to measure the additional processing time required by three sets of concurrent users, respectively, 10, 25, and 50. Each class executed, in a round-robin fashion, a set of five queries concurrently over 100 trials. The average length of the five different queries was 639 bytes, and the random key length was 32 bytes. The sample customer database created during the implementation was the target of the queries. The database, proxy, and client program were on separate $x86$ machines running RedHat Linux, within the same network. The overhead of proxy processing ranged from 183 to 316 μs for 10 to 50 concurrent users, respectively. Table 4 shows the proxy's performance.

TABLE 4
Proxy Overhead (in Microseconds)

Users	Min	Max	Mean	Std
10	74	1300	183.5	126.9
25	73	2782	223.8	268.1
50	73	6533	316.6	548.8

The worst-case scenario adds approximately 6.5 ms to the processing time of each query. Since acceptable response times for most Web applications usually fall between a few seconds to tens of seconds, depending on the purpose of the application, the additional processing time of the proxy contributes insignificant overhead in a majority of cases.

5 Further Discussion

5.1 Advantages

Randomizing network services and scripts not only hardens an individual system against code-injection attacks but also minimizes the possibility of network worms spreading by exploiting the same vulnerability against a popular software package: such malicious code will have to "guess" the correct key. As we saw in Section 2, the length of the key depends on certain architectural characteristics of the underlying processor and is typically much shorter than cryptographic keys. Nonetheless, the workload for a worm can increase by 2^{16} to 2^{32}, or even more. Periodically rerandomizing programs (e.g., when the system is recompiled for open-source operating systems, or at installation time and then periodically by an automated script for binary-only distributions) will further minimize the risk of persistent guessing attacks, as discussed in Section 2.4.5.

Compared to other protection techniques, our approach offers greater transparency to applications, languages, and compilers, none of which need to be modified (but see Section 5.2), and better performance at a fairly low complexity. Based on the ease with which we implemented the necessary extensions on the *bochs* emulator, we speculate that designers of new processors could easily include the appropriate circuitry. Since security is becoming increasingly important, adding security features in processors is seen as a way to increase market penetration. Various hardware manufacturers (in particular, Via) already has some provisions for cryptographic functionality embedded inside the processor, and the latest Crusoe processor includes a DES encryption engine. Although these seem to have been designed with digital rights management applications in mind, it may be possible to use the same mechanisms to enhance security in a different

application domain. However, using a full-featured block cipher such as DES or AES in our system is likely to prove too expensive, even if it is implemented inside the processor.

5.2 Disadvantages

Hardware support. Perhaps the main drawback of our approach as applied to binary code that is meant to execute on a hardware processor is the need for special support by the processor. In some programmable processors, it is possible to introduce such functionality in already-deployed systems. However, the vast majority of current processors do not allow for such flexibility. Thus, we are considering a more general approach of randomizing software as a way of introducing enough diversity among different instances of the same version of a piece of popular software that large-scale exploitation of vulnerabilities becomes infeasible. We view that work, which is still in progress, as complementary to the work we presented in this paper.

Static linking. A second drawback of our approach is that applications have to be statically linked, thus increasing their size. In our prototype of Section 2.4, we worked around this issue by using a single-image version of OpenBSD. In practice, we would seek to randomize (and thus statically link) only those programs that are exposed to remote exploits, i.e., network daemons, thus minimizing the overall impact of static linking to the system. Furthermore, it must be noted that avoiding static linking is going to help reduce only the disk usage, not the runtime memory requirements. Each randomized process will need to acquire (as part of process loading) and maintain its own copy of the randomized libraries using either of the following two mechanisms, neither of which is desirable. First, the process loader can load just the main program image initially, and dynamically copy/load and randomize libraries *on-demand.* This will incur considerable runtime overhead and also require complex process management logic, in the absence of which it will degrade to the other mechanism, described next. The second approach is to load and randomize the program code, and all libraries that are referenced, right at the beginning. It is obvious that this will result in large amounts of physical RAM being used to store multiple copies of the same code, some of which may never be executed during the entire life of the process.

Debugging. Debugging is made more difficult by the randomization process, since the debugger must be aware of it. The most straightforward solution is to derandomize the executable prior to debugging; the debugger can do this automatically, since the secret key is embedded in the ELF executable. Similarly, the debugger can use the key to derandomize core dumps.

Self-modifying code. Also note that our form of code randomization effectively precludes polymorphic and self-modifying code (both binary and scripts), as the randomization key and relevant code would then have to be encoded inside the program itself, potentially allowing an attacker to use them.

Since it is virtually impossible to distinguish between authorized and unauthorized memory writes, our architecture does not accommodate writes to the part of memory containing executable code or the ability of programs to dynamically generate (and execute) code. If the process does so, it will be corrupted, since the processor will write native instructions and then attempt to decode them as randomized instructions when attempting to execute them. One possibility for accommodating self-polymorphism is to require some form of IPC to a polymorphism service running as a separate process (or kernel service), which either returns properly randomized code or directly places such code in the caller's address space. However, it seems inevitable that any such approach will decrease overall system security.

Other dynamic-code constructs such as trampoline functions, which are injected by the operating system kernel for signal-handling purposes, are randomized on the fly by the operating system itself based on the randomization value stored in the PCB.

Code introspection. Our approach also prevents programs for performing code introspection, since instruction-fetch and data-read operations are handled by different data paths and logic in modern CPUs. Since our approach only modifies the instruction-fetch logic, a program reading its own code will see the encoded bits. This may also be a problem if an attacker manages to somehow read (parts of) the code space, since this may allow key extraction (depending on the randomization algorithm). One solution is for the operating system to mark code pages such that data reads (but not writes) from the code segment(s) return derandomized data (i.e., the unobfuscated instructions). This can increase the complexity of the on-processor logic and requires additional changes to the OS kernel.

Self-inflicted DoS. Instruction randomization should be viewed as a self-destruct mechanism: the program under attack is likely to go out of control and be terminated by the runtime environment. Thus, this technique cannot protect against DoS attacks and should be considered as a safe-guard of last resort, i.e., it should be used in conjunction with other techniques that prevent vulnerabilities leading to code-injection attacks from occurring in the first place. One such approach [26] uses an anomaly detection system coupled with ISR-protected software to allow the detector to "learn" attacks that trip the system and eventually filter them. Other similar work seeks to create input signatures for blocking future attacks [45], [23].

Comprehensive security. Finally, our approach does not protect against all types of buffer overflow attacks. In particular, overflows that only modify the contents of variables in the stack or the heap and cause changes in the control flow or logical operation of the program cannot be defended against using randomization. Similarly, our scheme does not protect against attacks that cause bad data to propagate in the system, e.g., not checking for certain Unix shell characters on input that is passed to the *system()* call. None of the systems we overview in Section 6 protect against the latter, and very few can deter the former type of attack. A more insidious overflow attack would transfer the control flow to some library function (*e.g., system()*). To defend against this, we propose to combine our scheme with randomizing the layout of code in memory at the granularity of individual functions [6], thus denying to an attacker the ability to jump to an already-known location in existing code. Although key-extraction attacks have been demonstrated against straightforward address-space randomization [35], a more comprehensive scheme has been shown to be resistant to such attacks [7].

6 Related Work

Encrypted software has been proposed in the context of software piracy and digital rights management. Such approaches treat code as content and consider users part of the threat model. Most of these approaches focused on the complex task of key management in such an environment (typically requiring custom-made processors with a built-in decryption engine such as DES). Our requirements for the randomization process are much more modest, allowing us to implement it on certain modern processors, emulators, and interpreters. We can take advantage of functionality in processors that allow for encrypted software (e.g., the Transmeta Crusoe TM5800 processor).

The work closest to ours is RISE [4], [5], which applies a randomization technique similar to ours for binary code only, and uses an emulator attached to specific processes (as opposed to a full system emulator). The inherent use of and dependency on emulation makes RISE simultaneously more practical for immediate use and inherently slower in the absence of hardware. Linn and Debray [24] use more general code obfuscation techniques to harden program binaries against static disassembly.

Feedback Learning IPS (FLIPS) [26] incorporates a supervision framework in the presence of suspicious traffic and uses ISR to isolate attack vectors, which are used to train the anomaly detector. A similar idea (fault-driven analysis to create signatures) using address-space obfuscation (see below) was independently proposed by Liang and Sekar [23] and Xu et al. [45]. Cox et al. [16] used artificial diversity to create quorum systems that make specific attacks impossible to successfully carry out.

PointGuard [15] encrypts all pointers while they reside in memory and decrypts them only before they are loaded to a CPU register. This is implemented as an extension to the GCC compiler, which injects the necessary instructions at compilation time, allowing a pure-software implementation of the scheme. Tuck et al. [40] discuss the use of hardware support in PointGuard for efficient code-pointer encryption to address read/write attacks, which allow an attacker to see snapshots of parts of a PointGuard-protected program's memory that enable future attacks. Xu [43] also proposes the use of control-data randomization to prevent attacks, focusing on compiler extensions to obfuscate function pointers and return addresses. Chen et al. [12] proposes the use of tainting in conjunction with hardware extensions to the processor, to detect compromised control-flow information.

Suh et al. [39] propose a hardware-based solution that can be used to thwart control-transfer attacks and restrict executable instructions by monitoring "tainted" input data. In order to identify "tainted" data, they rely on the operating system. If the processor detects the use of this tainted data as a jump address or an executed instruction, it raises an exception that can be handled by the operating system. DIRA [37] is a technique for automatic detection, identification, and repair of control-hijacking attacks. This solution is implemented as a GCC compiler extension that transforms a program's source code adding heavy instrumentation so that the resulting program can perform these tasks. The use of checkpoints throughout the program ensures that corruption of state can be detected if control sensitive data structures are overwritten. TaintCheck [28] performs dynamic taint analysis to detect overwrite attacks. TaintCheck is implemented as an extension to Valgrind. Although it does not require modifications to the source code of a monitored program, it imposes a performance penalty typical of emulator-based approaches and requires memory resources to store taint information.

Abadi et al. [1] formalized the concept of Control Flow Integrity (CFI), observing that high-level programming often assumes properties of control flow that are not enforced at the machine level. CFI provides a way to statically verify that execution proceeds within a given control-flow graph (the CFG effectively serves as a policy). CFI enables the efficient implementation of a software shadow call stack with strong protection guarantees.

Another approach, address obfuscation [6], [29], randomizes the absolute locations of all code and data, as well as the distances between different data items. Several transformations are used, such as randomizing the base addresses of memory regions (stack, heap, dynamically linked libraries, routines, static data, and so forth), permuting the order of variables/routines, and introducing random gaps between objects. Although very effective against *jump-into-libc* attacks, it is less so against other common attacks, due to the fact that the amount of possible randomization is relatively small. However, address obfuscation can protect against attacks that aim to corrupt variables or other data. This approach can be effectively combined with instruction randomization to offer comprehensive protection against all memory-corrupting attacks. A similar approach was independently proposed by Xu et al. [44]. Shacham et al. [35] show that in many cases address-space randomization is not very effective, because 32-bit address spaces do not allow for sufficient entropy in the placement of data and control information in memory. However, Bhatkar et al. [7] argue that it is possible to introduce enough entropy in address-space obfuscation by expanding the scope of randomization to include relative code placement at the granularity of individual functions, static- and stack-resident data, and to continuously rerandomize these at runtime. Sovarel et al. [38] demonstrate a key-extraction attack against ISR systems when the system always uses the same key across program crashes due to the attack.

Hardware features such as the NX flag in recent Pentium-class processors [19] or other similar proposals [47] address the performance issue but only cover a subset of exploitation methods (e.g., jump-into-libc attacks are still possible); note that ISR is also susceptible to such attacks. Lam and Chiueh [22] use the $x86$ segmentation features to implement array bounds checking for most of the buffers in a program, resulting in relatively low protection-mechanism overhead.

SQL injection. It is becoming common practice to use PREPARE statements, which allow a client to preissue a template SQL query at the beginning of a session; for the actual queries, the client only needs to specify the variables that change. Although the PREPARE feature was introduced as a performance optimization, it can address SQL injection attacks if the same query is issued many times. When the queries are dynamically constructed, this approach does not work well.

Pietraszek and Berghe [31] proposed the use of taint analysis to detect SQL injection attacks, and a similar system is described by Xu et al. [46]. While this approach requires a much tighter coupling between the query composer (e.g., the CGI script) and the consumer (the SQL

parser in the database), it is conceptually similar to our view that commands should originate from the program text and not be derived from untrusted data.

AMNESIA [20] uses static analysis and training to build query models. These models, expressed as character-level automata, are used at runtime to monitor and enforce compliance of the issued queries. Enforcement is done by adding checkpoints in the application source code. Possible limitations include false positives and false negatives, although the experiments performed did not reveal any such problems in practice.

7 Conclusions

We have described our ISR scheme for countering code-injection attacks. We protect against **any** type of code-injection attacks by creating an execution environment that is unique to the running process. Injected code will be invalid for that execution environment and, thus, cause an exception. This approach is equally applicable to machine-code executables and interpreted code. To evaluate the feasibility of ISR, we constructed three prototypes, one each for $x86$ machine code, Perl, and SQL.

The ease of implementation in all cases leads us to believe that our approach is feasible in hardware and offers significant benefits in terms of transparency and performance, compared to previously proposed techniques. Furthermore, the operating system modifications were minimal, making this an easy feature to support. In the case of SQLrand, we showed that we can achieve portability and security gains through using a proxy-based implementation, while incurring a minimal performance overhead of at most 6.5 ms per query.

Admittedly, our solution does not address the core issue of software vulnerabilities, which is the bad quality of code. Given the apparent resistance to the wide adoption of safe languages, and not foreseeing any improvement in programming practices in the near future, we believe our approach can play a significant role in hardening systems and invalidating the "write-once exploit-everywhere" principle of software exploits.

Acknowledgments

This work was supported in part by the US National Science Foundation under Grant ITR CNS-04-26623 and Grant ANI-0133537.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity: Principles, Implementations, and Applications," *Proc. 12th ACM Conf. Computer and Comm. Security (CCS '05),* Nov. 2005.

[2] Aleph One, "Smashing the Stack for Fun and Profit," *Phrack,* vol. 7, no. 49, 1996.

[3] C. Anley, *Advanced SQL Injection in SQL Server Applications,* 2008.

[4] E.G. Barrantes, D.H. Ackley, S. Forrest, T.S. Palmer, D. Stefanovic, and D.D. Zovi, "Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks," *Proc. 10th ACM Conf. Computer and Comm. Security (CCS '03),* pp. 281-289, Oct. 2003.

[5] E.G. Barrantes, D.H. Ackley, S. Forrest, and D. Stefanovic, "Randomized Instruction Set Emulation," *ACM Trans. Information and System Security,* vol. 8, no. 1, pp. 3-40, Feb. 2005.

[6] S. Bhatkar, D.C. DuVarney, and R. Sekar, "Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits," *Proc. 12th USENIX Security Symp.,* pp. 105-120, Aug. 2003.

[7] S. Bhatkar, R. Sekar, and D.C. DuVarney, "Efficient Techniques for Comprehensive Protection from Memory Error Exploits," *Proc. 14th USENIX Security Symp.,* pp. 255-270, Aug. 2005.

[8] *Bochs Emulator Web Page,* http://bochs.sourceforge.net/, 2008.

[9] D. Bruening, T. Garnett, and S. Amarasinghe, "An Infrastructure for Adaptive Dynamic Optimization," *Proc. Symp. Code Generation and Optimization (CGO '03),* pp. 265-275, 2003.

[10] *CERT Vulnerability Note VU#496064,* http://www.kb.cert.org/vuls/id/496064, Apr. 2002.

[11] *CERT Vulnerability Note VU#282403,* http://www.kb.cert.org/vuls/id/282403, Sept. 2002.

[12] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and C. Verbowski, "Defeating Memory Corruption Attacks via Pointer Taintedness Detection," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '05),* pp. 378-387, June 2005.

[13] S. Chen, J. Xu, E.C. Sezer, P. Gauriar, and R.K. Iyer, "Non-Control-Data Attacks Are Realistic Threats," *Proc. 14th USENIX Security Symp.,* pp. 177-191, Aug. 2005.

[14] M. Costa, J. Crowcroft, M. Castro, and A. Rowstron, "Vigilante: End-to-End Containment of Internet Worms," *Proc. 20th Symp. Systems and Operating Systems Principles (SOSP),* 2005.

[15] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities," *Proc. 12th USENIX Security Symp.,* pp. 91-104, Aug. 2003.

[16] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "*N*-Variant Systems: A Secretless Framework for Security through Diversity," *Proc. 15th USENIX Security Symp.,* pp. 105-120, July/Aug. 2005.

[17] G.W. Dunlap, S.T. King, S. Cinar, M.A. Basrai, and P.M. Chen, "ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay," *Proc. Fifth Symp. Operating Systems Design and Implementation (OSDI '02),* Dec. 2002.

[18] D. Evans and D. Larochelle, "Improving Security Using Extensible Lightweight Static Analysis," *IEEE Software,* Jan./Feb. 2002.

[19] L. Garber, "New Chips Stop Buffer Overflow Attacks," *Computer,* vol. 37, no. 10, p. 28, Oct. 2004.

[20] W.G.J. Halfond and A. Orso, "SQL Command-Form Coverage for Testing Database Applications," *Proc. 20th Int'l Conf. Automated Software Eng. (ASE '05),* Sept. 2005.

[21] G.S. Kc, A.D. Keromytis, and V. Prevelakis, "Countering Code-Injection Attacks with Instruction-Set Randomization," *Proc. 10th ACM Conf. Computer and Comm. Security (CCS '03),* Oct. 2003.

[22] L. Lam and T. Chiueh, "Checking Array Bound Violation Using Segmentation Hardware," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '05),* pp. 388-397, June 2005.

[23] Z. Liang and R. Sekar, "Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers," *Proc. 12th ACM Conf. Computer and Comm. Security (CCS '05),* pp. 213-222, Nov. 2005.

[24] C. Linn and S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," *Proc. 10th ACM Conf. Computer and Comm. Security (CCS '03),* pp. 290-299, Oct. 2003.

[25] D. Litchfield, *Web Application Disassembly with ODBC Error Messages,* http://www.nextgenss.com/papers/webappdis.doc, 2008.

[26] M. Locasto, K. Wang, A. Keromytis, and S. Stolfo, "FLIPS: Hybrid Adaptive Intrusion Prevention," *Proc. Eighth Symp. Recent Advances in Intrusion Detection (RAID '05),* pp. 82-101, Sept. 2005.

[27] M. Conover and w00w00 Security Team, *w00w00 on Heap Overflows,* http://www.w00w00.org/files/articles/heaptut.txt, 2008.

[28] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," *Proc. 12th Ann. Symp. Network and Distributed System Security (SNDSS '05),* Feb. 2005.

[29] *PaX Home Page,* http://pax.grsecurity.net/, 2008.

[30] *Perltidy Home Page,* http://perltidy.sourceforge.net/, 2008.

[31] T. Pietraszek and C.V. Berghe, "Defending against Injection Attacks through Context-Sensitive String Evaluation," *Proc. Eighth Int'l Symp. Recent Advances in Intrusion Detection (RAID '05),* Sept. 2005.

[32] J. Pincus and B. Baker, "Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overflows," *IEEE Security and Privacy Magazine,* vol. 2, no. 4, pp. 20-27, July/Aug. 2004.

[33] V. Prevelakis and A.D. Keromytis, "Drop-in Security for Distributed and Portable Computing Elements," *Internet Research: Electronic Networking, Applications and Policy,* vol. 13, no. 2, 2003.

[34] B. Rogers, Y. Solihin, and M. Prvulovic, "Memory Predecryption: Hiding the Latency Overhead of Memory Encryption," *Proc. Workshop Architectural Support for Security and Anti-Virus (WASSA '04),* pp. 22-28, Oct. 2004.

[35] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh, "On the Effectiveness of Address-Space Randomization," *Proc. 11th ACM Conf. Computer and Comm. Security (CCS '04),* pp. 298-307, Oct. 2004.

[36] S. Sidiroglou and A.D. Keromytis, "A Network Worm Vaccine Architecture," *Proc. IEEE Int'l Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE '03), Workshop Enterprise Security,* pp. 220-225, June 2003.

[37] A. Smirnov and T. Chiueh, "DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks," *Proc. ISOC Symp. Network and Distributed System Security (SNDSS '05),* Feb. 2005.

[38] A.N. Sovarel, D. Evans, and N. Paul, "Where's the FEEB? The Effectiveness of Instruction Set Randomization," *Proc. 14th USENIX Security Symp.,* pp. 145-160, Aug. 2005.

[39] G.E. Suh, J.W. Lee, D. Zhang, and S. Devadas, "Secure Program Execution via Dynamic Information Flow Tracking," *SIGOPS Operating Systems Rev.,* vol. 38, no. 5, pp. 85-96, 2004.

[40] N. Tuck, B. Calder, and G. Varghese, "Hardware and Binary Modification Support for Code Pointer Protection from Buffer Overflow," *Proc. 37th Int'l Symp. Microarchitecture (MICRO '04),* pp. 209-220, Dec. 2004.

[41] D. Wagner, J.S. Foster, E.A. Brewer, and A. Aiken, "A First Step towards Automated Detection of Buffer Overrun Vulnerabilities," *Proc. ISOC Symp. Network and Distributed System Security (SNDSS '00),* pp. 3-17, Feb. 2000.

[42] A. Whitaker, M. Shaw, and S.D. Gribble, "Scale and Performance in the Denali Isolation Kernel," *Proc. Fifth Symp. Operating Systems Design and Implementation (OSDI '02),* Dec. 2002.

[43] J. Xu, "Intrusion Prevention Using Control Data Randomization," *Proc. IEEE Int'l Conf. Dependable Systems and Networks (DSN '03),* June 2003.

[44] J. Xu, Z. Kalbarczyk, and R.K. Iyer, "Transparent Runtime Randomization for Security," *Proc. 22nd Int'l Symp. Reliable Distributed Systems (SRDS '03),* pp. 260-273, Oct. 2003.

[45] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt, "Automatic Diagnosis and Response to Memory Corruption Vulnerabilities," *Proc. 12th ACM Conf. Computer and Comm. Security (CCS '05),* pp. 222-234, Nov. 2005.

[46] W. Xu, S. Bhatkar, and R. Sekar, "Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks," *Proc. USENIX Security Symp.,* pp. 121-136, July/Aug. 2006.

[47] D. Ye and D. Kaeli, "A Reliable Return Address Stack: Microarchitectural Features to Defeat Stack Smashing," *Proc. Workshop Architectural Support for Security and Anti-Virus (WASSA '04),* pp. 69-76, Oct. 2004.

Stephen W. Boyd is a principal software developer in the Fraud Management Division, SAS Institute.

Gaurav S. Kc received the BEng (Hons) degree in computing from Imperial College of Science, Technology and Medicine, London, in 1999 and the MS, MPhil, and PhD degrees in computer science from Columbia University in 2001, 2003, and 2005, respectively. In 2005, he joined Google, where he is a software engineer working on computer security issues in Google's production infrastructure.

Michael E. Locasto is an Institute for Information Infrastructure Protection (I3P) fellow and a visiting research assistant professor in the Department of Computer Science, George Mason University. He seeks to understand why it seems difficult to build secure systems and how we can get better at it. His research explores methods for applying machine intelligence to a variety of security mechanisms, especially ways to make intrusion defense systems automatic, correct, and adaptive.

Angelos D. Keromytis received the BSc degree in computer science from the University of Crete, Greece, and the MSc and PhD degrees from the University of Pennsylvania. He is an associate professor in the Department of Computer Science, Columbia University and the director of the Network Security Laboratory. His research interests revolve around systems and network security.

Vassilis Prevelakis received the BSc and MSc degrees from the University of Kent, Canterbury, United Kingdom and the PhD degree from the University of Geneva, Switzerland. He is the director of the AEGIS Research Center in Information Security, Athens. He has worked in various areas of security in systems and networks both as an academic and as a freelance consultant. His current research involves issues related to automation network security, secure software design, autoconfiguration issues in secure VPNs, and so forth. He has published numerous papers in these areas and is actively involved in standards bodies such as the IETF. He has received research funding from Defense Advanced Research Projects Agency (DARPA CHATS) and from US National Science Foundation (NSF CAREER).

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.