Global ISR: Toward a Comprehensive Defense Against Unauthorized Code Execution

Georgios Portokalidis and Angelos D. Keromytis

Network Security Lab, Columbia University, New York, NY 10027, USA
{porto, angelos}@cs.columbia.edu

Abstract. Instruction-set randomization (ISR) obfuscates the "language" understood by a system to protect against code-injection attacks by presenting an ever-changing target. ISR was originally motivated by code injection through buffer overflow vulnerabilities. However, Stuxnet demonstrated that attackers can exploit other vectors to place malicious binaries into a victim's filesystem and successfully launch them, bypassing most mechanisms proposed to counter buffer overflows. We propose the holistic adoption of ISR across the software stack, preventing the execution of unauthorized binaries and scripts regardless of their origin. Our approach requires that programs be randomized with different keys during a user-controlled installation, effectively combining the benefits of code whitelisting/signing and runtime program integrity. We discuss how an ISR-enabled environment for binaries can be implemented with little overhead in hardware, and show that higher-overhead software-only alternatives are possible. We use Perl and SQL to demonstrate the application of ISR in scripting environments with negligible overhead.

1 Introduction

Code-injection attacks occur when an attacker implants arbitrary code into a vulnerable program to gain unauthorized access on a system, acquire elevated privileges, or extract sensitive information like passwords. In the past, code-injection (CI) attacks accounted for almost half of the advisories released by CERT [1]. Because such attacks can be launched over the network, they were regularly used as an infection vector by many computer worms [2–6]. More recently, they have been outweighed by other types of attacks, such as SQL injection and Cross Site Scripting (XSS). However, the recent Conficker [7] and Stuxnet [8] worm outbreaks, and the multiple vulnerabilities discovered in Adobe's omnipresent software [9,10], serve as a reminder that CI attacks still pose a significant threat to a large number of systems.

Code-injection attacks are frequently enabled by vulnerabilities such as buffer overflows [11–13], and other memory corruption vulnerabilities like dangling pointers [14, 15] and format string attacks [16], which allow attackers to first inject code in the vulnerable program, and then redirect its execution to the injected code. In their simplest form, they follow the overflow of a buffer in the

stack, which overwrites the function's return address to transfer control to the code placed in the buffer as part of the overflow.

The techniques used to perform CI attacks can vary significantly. As such, one could assume that to stop these attacks, we would need to concurrently prevent all the types of exploits that make them possible. However, while the specific techniques used in each attack differ, they all result in attackers executing their code. This capability implies that attackers know what "type" of code (*e.g.,* x86) is understood by the system.

This observation led us [17] (and concurrently others [18,19]) to introduce a general approach for preventing code-injection attacks, *instruction-set randomization (ISR).* Inspired by biology where genetic variation protects organisms from environmental threats, ISR proposes the randomization of the underlying system's instructions, so that "foreign" code introduced within a running process by an attack would fail to execute correctly, regardless of the injection approach. ISR is a general approach that defeats *all types* of remote code injection, regardless of the way it was injected into a process, by providing an ever-shifting target to the attacker.

ISR protects running processes against all types of CI attacks, but it cannot protect against the most basic type of attack. Consider the case where the attacker manages to get his own malicious binary on the victim's file system, and then finds some way to launch that binary. ISR, as defined in previous work, would either randomize and then execute this binary, or would simply run it as-is since it is not randomized. In both cases the attacker can completely bypass ISR. However, the same holds for almost all other protection techniques aimed at the same class of vulnerabilities, such as address space layout randomization (ASLR). Comprehensive techniques such as Taint Tracking suffer from low performance and complexity, due to the requirement for continuous and complete monitoring of all that occurs within a system.

The means by which the attacker gets his malicious program (frequently called malware) on the target's system is secondary. Sophisticated attackers exploit other vulnerabilities to copy the malware on the victim's file system and then execute it, or overwrite already existing binaries, so that the next time the user launches the application the attacker's code executes instead. For instance, consider the Stuxnet [8] worm, which is considered one of the most sophisticated ever encountered, and for this reason it has received a lot of attention both from researchers and the media. Its main goal has been identified as being the infection of industrial control systems (ICS), but to propagate and reach its target, it uses many elaborately constructed infection vectors. One of its propagation methods involves a code-injection attack that follows the exploitation of a buffer overflow vulnerability in the Windows Server Service. ISR would avert the exploitation of this vulnerability, effectively disabling this infection vector. Unfortunately, Stuxnet can evade ISR by exploiting other vulnerabilities to place a binary in the target's file system (both remotely, and locally through removable storage), and then execute it.

We propose the whole-sale adoption of ISR across all system layers, as a way to enforce non-von Neumann semantics. Doing so will prevent the execution of unauthorized code, and will protect systems from complex threats such as Stuxnet. Our approach requires that all binaries in the system are pre-randomized. Our approach requires that all binaries in the system are pre-randomized. New binaries being installed will require user authorization before being copied to the filesystem, which will also randomize them in the process. As a result an unauthorized binary being dropped on the filesystem will not be able to execute, as it will be in the wrong "language". The benefits we obtain with this approach are twofold. First, we ensure the integrity of existing system binaries, and at the same time we ensure that new binaries that have not been installed through the proper channels (*e.g.,* an installer tool with proper rights) will fail to execute successfully. Thus, ISR simultaneously provides the benefits of program signing/whitelisting and runtime integrity.

Employing ISR in this fashion would thwart the propagation of worms like Stuxnet. Future worms would likely then resort in injecting interpreted code, if an interpreter for a language like Perl or Python is available. Injecting interpreted code would enable the attacker to bypass binary ISR and execute code, but it could be also used to extract information from the system that would enable him to "break" the randomization used (*e.g.,* by guessing the key used for randomizing authorized binaries). Fortunately, ISR is flexible enough that it can be also applied on interpreted languages like Perl and SQL, as we demonstrate in Sections 5 and 6.

In the remainder of this chapter, we discuss the principles behind ISR, our early work, and the open challenges in adopting it across the whole software stack.

2 Instruction-Set Randomization

ISR is based on the observation that code-injection attacks need to position executable code within the address space of the exploited application and then redirect control to it. The injected code needs to be compatible with the execution environment for these attacks to succeed. This means that the code needs to be compatible with the processor and software running at the target. For instance, injecting x86 code into a process running on an ARM system will most probably cause it to crash, either because of an illegal instruction being executed, or due to an illegal memory access. Note that for this particular example, it may be possible to construct somewhat limited code that will run without errors on both architectures.

We build on this observation to prevent attackers from executing injected code. We introduce a randomly mutating execution environment, whose "language" is not known to attackers, while legitimate binaries are "translated" to this language during installation. In this way, both injected code and binaries installed without authorization will fail to execute. Attempts to guess the language of the execution environment can be hindered by frequently mutating it,

and by allowing for different parts of a program to "speak" different languages. For instance, every time an application crashes, which could be due to a failed attack, we re-randomize it, while various components of the application (*i.e.,* libraries or even functions) can be randomized in a different way. Attempts to execute the code injected into a randomized process will still crash it causing a denial of service (DoS), but attackers will no longer be able to perform any useful actions such as installing rootkits.

The strength of ISR lies in the fact that the attacker does not know the instruction set used by an application, and the high complexity of guessing it. As such, if an attacker has access to the original code, and he can gain access to the randomized code, he can launch an attack against the applied transformation to attempt to learn the new instruction set. This requires that the attacker has local access to the target host. In general, ISR is primarily focused on protecting against remote attacks that originate from the network, where the attacker does not have access to the target system or the randomized binaries. Consequently, we assume that attackers do not have local access.

Finally, the security of the scheme depends on the assumption that the attacker's code will raise an exception (*e.g.,* by accessing an illegal address or using an invalid opcode), similarly to the example where x86 code is injected into an application for ARM. While this will be generally true, there are a few permutations of injected code that will result in working code that performs the attacker's task. We argue that this number will be statistically insignificant [19], and it is comparable with the probability of creating a valid buffer-overflow exploit using the output of a random number generator as code.

2.1 ISR Operation

CPU instructions for common architectures, like x86 and ARM, consist of two parts: the *opcode* and *operands*. The opcode defines the action to be performed, while the operands are the arguments. For example, in the x86 architecture a software interrupt instruction (*INT*) comprises of the opcode *0xCD*, followed by a one-byte operand that specifies the type of interrupt. We can create new instruction sets by randomly creating new mappings between opcodes and actions. We can further randomize the instruction set by also including the operands in the transformation.

For ISR to be effective and efficient, the number of possible instruction sets must be large, and the mapping between the new opcodes and instructions should be efficient (*i.e.,* not completely arbitrary). We can achieve both these properties by employing cryptographic algorithms and a randomly generated secret key. As an example, consider a generic RISC processor with fixed-length 32-bit instructions. We can effectively generate random instruction sets by encoding instructions with XOR and a secret 32-bit key. In this example, an attacker would have to try at most 2^{32} combinations to guess the key. Architectures with larger instructions (*i.e.,* 64 bits) can use longer keys to be even more resistant to brute-force attacks.

In the case of XOR, using a key size larger than the instruction size does not necessarily improve security, since the attacker may be able to attack it in a piece-meal fashion (*i.e.,* guess the bits corresponding to one instruction, and then proceed with guessing the bits for a second instruction in a sequence, and so forth). The situation is even more complicated on architectures with variable sized instructions like the x86. Many instructions in the x86 architecture are 1 or 2 bytes long. This effectively splits a 32-bit key in four or two sub-keys of 8 and 16 bits respectively. Thus, it is possible that a remote attacker attempts to guess each of the sub-keys independently [20]. A failed attempt to guess the key will cause the application to crash, which could be potentially detected by the attacker.

The deficiencies of XOR randomization on architectures like the x86 can be overcome using other ciphers that cannot be attacked in a piece-meal fashion. For example, using bit-transposition with a 32-bit instruction requires an 160-bit key. Although not all possible permutations are valid (the effective key size is $log_2(32!)$), the work factor for the attacker is high, as he would have to try at most 32! combinations to guess the key (notice that $32! >> 2^{32}$). 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 we must have simultaneous access to the whole block of instructions during execution. This increases complexity, specially when implementing ISR in hardware. If the block size is not an issue, such as in software-only implementations of ISR, we could also use AES encryption with 128-bit blocks of code.

We adopt a different approach to protect against key guessing attacks. First, we re-randomize an application every time it crashes using a new randomly generated key. Thus, an attacker trying to remotely guess the key being used, will cause it to change with each failed attempt. Additionally, we use different keys to randomize different parts of an application, where the execution environment allows it. For instance, we can randomize every library or even function used by an application with a different key.

2.2 Randomization of Binaries

ELF (the executable and linking format) is a very common and standard file format used for executables and shared libraries in many Unix-type systems like Linux, BSD, Solaris, *etc.* Despite the fact that it is most commonly found on Unix systems, it is very flexible, and it is not bound to any particular architecture or OS. Additionally, the ELF format completely separates code and data, including control data such as the procedure linkage table (PLT), making binary randomization straightforward.

We modified the *objcopy* utility, which is part of the GNU binutils package to add support for randomizing ELF executables and libraries. *objcopy* can be used to perform certain transformations (*e.g.,* strip debugging symbols) on an object file, or simply copy it to another. Thus, it is able to parse the ELF headers of an executable or library and access its code. We modified *objcopy* to randomize a file's code using XOR and one or more 16-bit keys. Using a 16-bit key is sufficient

as an attacker has an $1/2^{16}$ chance to correctly guess the key in one attempt, while failed guess attempts will cause the application to crash, and consequently trigger the re-randomization of the binary with a different key. Multiple keys can be used to randomize a single binary, in the case of software-only ISR. When using multiple keys, each function within the binary is encoded using a different key. As this will greatly affect the number of keys being used in the system, we expect such a setup to be only used with highly critical applications.

The keys for every binary in the system are stored in a database (DB), using the *sqlite* database system. *Sqlite* is a software library that implements a self-contained, serverless SQL database engine. The entire DB is stored in a single file, and can be accessed directly by the loader (using the *sqlite* library) without the need to run additional servers. Keys are indexed using a binary's full path on disk, and the operation of retrieving them from the DB is fast. Since it is an operation that is only performed when a binary is loaded to memory (*e.g.,* when an application is launched or a dynamic library is loaded), its performance is not critical for the system. New binaries can be installed using an installation script, which uses our modified version of *objcopy* to randomize the binaries being installed to the file system. Authorization is requested before completing the installation to copy the files, and insert the randomization keys in the DB.

Note that randomizing using XOR does not require that the target binary is aligned, so it does not increase its size or modify its layout. Moreover, while our implementation is currently only able to randomize ELF binaries, support for other binaries can be easily added. For instance, we plan to extend *objcopy* to also randomize *Portable Executable (PE)* binaries for Windows operating systems [21].

2.3 Execution Environment

A randomized process requires the appropriate execution environment to obtain the keys used by an applications and its libraries, and to de-randomize its instructions before executing them. Such an execution environment can be implemented both in hardware (Section 3) and software (Section 4).

Additionally, when an application is using multiple keys to randomize its various parts, the execution environment needs to be able to detect when execution switches between parts of the code using different randomization keys. For instance, each library being used by an application may be randomized with a different key, and even different functions within the application and libraries can be randomized with varying keys. Detecting such context switches can be complex in hardware, and as we discuss below, the proposed hardware-based implementation of ISR only handles statically linked executables (sacrificing flexibility for performance). In Section 4, we describe a software-only ISR solution that handles dynamically linked applications, and supports multiple instruction sets per process (*i.e.,* instructions randomized with different keys), albeit with higher overhead for the randomized applications.

3 Hardware-based ISR

Implementing ISR in hardware requires a programmable processor [22] or small modifications to current processor architectures like the IA-32 to perform the de-randomization of instructions before execution. Let us consider such a system running on top of such a CPU. Typically, software is separated between kernel and user space, where tasks such as virtual memory management and device drivers are running in *kernel space*, and user processes in *user space*. ISR aims to protect applications, and as such we currently ignore kernel space, which also simplifies our design because we do not have to consider the interactions between ISR and the various low-level processor events (*e.g.,* interrupts). However, a comprehensive solution would also apply ISR within the kernel.

When a new randomized process is launched (*e.g.,* as a result of an *exec()* system call), the processor needs to know the key being used before executing any of its instructions. The key used to randomize the binary is stored in the *sqlite* DB, which is stored in a file (as described in Section 2.2). We query the DB for the binary's key through a user space component, and then store it in the process' process control block (PCB) structure. When the process is actually scheduled for execution, the OS loads the key in the PCB on the processor. For this purpose, we provide for a special processor register where the decoding key is stored, and a special privileged instruction (called *GAVL*) that allows write-only access to this register when running in privileged mode (*i.e.,* in kernel space). To accommodate code that is not randomized (*e.g.,* the kernel and *init*), we provide a special key that, when loaded via the *GAVL* instruction, disables the decoding process. 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 a program is executed by the processor, instructions are first fetched from memory, decoded (this refers to the CPU's decoding and not ISR's), and then executed. Our design introduces a de-randomizing element, which lies between the fetching of the instruction, and its decoding. This element is responsible for de-randomizing the code fetched from memory, before delivering it to the CPU for decoding and execution. Such a scheme can be very efficiently implemented in the interface between the L2 cache and main memory, as shown by Rogers *et al.* [23]. When XOR randomization is used, this element simply applies XOR on the bytes received from memory using the key stored in *GAVL*.

Normally, applications use various libraries, which can be linked statically, or dynamically during loading, or even at runtime. But the ISR key is associated with an entire process, making it difficult to accommodate dynamically linked libraries. We could require that all shared libraries used by an application are randomized with its key, but then the memory occupied by each library will not be actually shared with other processes, as they may use a different key. To keep the hardware design simple and efficient, our early prototypes required that applications running under ISR are statically linked. Moreover, modern Linux systems frequently include a read-only *virtual shared object (VDSO)* in every running process. This object is used to export certain kernel functions to

user space. For instance, it is used to perform system calls, replacing the older software interrupt mechanism (*INT 0x80*). If the use of a VDSO is required, we need to make small modifications to the kernel, so that a unique non-shared object is mapped to each process, and to randomize it using the process' key.

Finally, there are cases where the kernel injects a few non-randomized instructions in processes. For instance, some systems inject code within the stack of a process when a signal is delivered. These *signal trampolines* are used to set and clean up the context of signal handlers. They are a type of legitimate code-injection (approximately 5-7 instructions long) performed by the system itself. Fortunately, since the kernel performs this code-injection, we can modify it to randomize these instructions before injecting them in a process.

3.1 x86 Prototype Using Bochs

To determine the feasibility of a hardware-based ISR implementation, we built a prototype for the most widely used processor architecture, the x86, using the Bochs emulator [24]. As we discussed in Section 2, randomization on the x86 is more complicated than with RISC-type processors, because of its variable-sized instructions. So, by implementing our prototype for x86, we also test the feasibility of ISR in a worst-case scenario.

Bochs is an open-source emulator of the x86 architecture, which operates by interpreting every machine instruction in software. Bochs, in many ways, operates similarly to real hardware. For instance, in its core we find the CPU execution loop, which calls the function *fetchDecode()* that fetches an instruction from the emulator's virtual RAM, and decodes it. This behavior closely simulates the i486 and Pentium processors, with their instruction *prefetch streaming buffer*, which keeps the next 16 bytes of instructions (32 bytes on later processors). We implemented the de-randomization element in the beginning of *fetchDecode()*, after the fetching of an instruction byte, and before the decoding, as we would do in the real hardware. The decoding is driven by the contents of the *GAVL* register, which if empty indicates that instructions are not randomized, while otherwise contains the decoding key.

To simplify the creation and evaluation of our prototype, we adopted the techniques we used to construct embedded systems for VPN gateways [25]. We use automated scripts to produce compact (2-4MB) 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.

	ftp	**sendmail**	**fibonacci**
Bochs	39.0s	$\approx$ 28s	$\approx 93s$
Native	29.2s	$\approx$ 1.35s	0.322s

Table 1. Average execution times (in seconds) for identical binaries under Bochs, and native execution (on the same host). The performance numbers of individual runs were within 10% of the listed averages.

Although this greatly limits the real-world applicability of our prototype, we feel it is an acceptable compromise for evaluating hardware-based ISR. Section 4 describes in detail a more practical software-only implementation of ISR.

The root of the runtime file system, together with the executable and associated links, are 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 (*i.e.,* a file that Bochs treats as a virtual hard drive). This file system image also contains the */etc* directory of the running system in uncompressed form, which allows us to easily reconfigure the runtime parameters. At boot time, the kernel is expanded from the boot image to Bochs' main memory, and executed. The */etc* directory is then copied from the bootable medium to the RAM-disk, so that the entire system is running entirely off it. This organization allows multiple applications to be combined with a single kernel, while leaving the configuration files in the */etc* directory on the boot medium.

3.2 Performance

Our Bochs prototype only serves the purpose of demonstrating the feasibility of an ISR implementation in hardware. Generally, interpreting emulators (as opposed to virtual machine emulators, such as VMWare and VirtualBox) impose a considerable performance penalty that ranges from a slow-down of one to several orders of magnitude. This makes the direct application of ISR using such an emulator impractical for production software, 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, like the repeated upload and download of a $\approx$200KB file, creation, deletion and renaming of directories, and generating directory listings by means of an automated script. We repeated the same sequence of operations 10 times, and list the average. The results indicate that a network I/O-intensive process does not suffer execution time slowdown proportional to the reduction in processor speed. Next, the second column in the table shows the overall time needed by *sendmail* to receive 100 short e-mails of $\approx$1KB each from a remote host.

In contrast, the third column demonstrates the significant slowdown incurred by 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.

4 Software-only ISR

A fast implementation of ISR, built entirely in software, is currently the only way to apply ISR on production systems, since the idea of ISR-enabled hardware has had little allure with hardware vendors. Software-only implementations of ISR have been proposed before [26], but have seen little use in practice as they cannot be directly applied to commodity systems. For instance, they do not support shared libraries or dynamically loaded libraries (*i.e.,* they require that the application is statically linked), and increase the code size of encoded applications.

Our approach, much like previous solutions, uses dynamic binary translation to apply ISR on unmodified binaries, but it supports processes randomized with multiple keys (*e.g.,* shared libraries or even functions can use different keys), and incurs low overhead. Our tool builds on Intel's dynamic instrumentation tool called PIN [27], which provides the runtime environment. Similarly to hardware-based ISR, application code is randomized using the XOR function and a 16-bit key, and the application is re-randomized with a new key every time it crashes to make it resistant to remote key guessing attacks [20].

Supporting multiple keys per process means that every shared library used by a process can be randomized using a different key, and applications no longer need to be statically linked. When an application crashes, we do not need to re-randomize all the shared libraries used by it. Instead, we examine the key being used at the time of the crash, and re-randomize only the part of the process that was using that key, since the crash could not have revealed information about other keys to an attacker. Otherwise, we need to dynamically re-randomize the relevant libraries, and propagate the key to other processes that are concurrently using that library.

4.1 ISR Using PIN

We implemented the de-randomizing execution environment for x86 software running on Linux[1], using Intel's dynamic binary instrumentation tool PIN [27]. PIN is an extremely versatile tool that operates entirely in user space, and supports multiple architectures (x86, 64-bit x86, ARM) and operating systems (Linux, Windows, MacOS). It operates by just-in-time (JIT) compiling the target's instructions combined with any instrumentation into new code, which is placed into a code cache, and executed from there. It also offers a rich API to inspect and modify an application's original instructions.

We make use of the supplied API to implement our ISR-enabled runtime. First, we install a callback that intercepts the loading of all file images. This

[1] While the current implementation only works on Linux, it can be easily ported to other platforms also supported by the runtime

provides us with the names of all the shared libraries being used, and the memory ranges where they have been loaded in the address space. We use the path and name of a library to query the DB for the key or keys used by the library. We save the returned keys, along with the memory address ranges that they correspond to, in a hash table-like data structure that allows us to quickly look up a key using a memory address.

The actual de-randomization is performed by installing a callback that replaces PIN's default function for fetching code from the target process. This second callback reads instructions from memory, and uses the memory address to look up the key to use for decoding. To avoid performing a look up for every instruction fetched, we cache the last used key. During our evaluation this simple single entry cache achieved high hit ratios, so we did not explore other caching mechanisms. All instructions fetched from memory that have not been associated with a key are considered to be part of the executable, and are decoded using its key.

Memory Protection (MP) When executing an application within PIN, they both operate on the same address space. This means that in theory an application can access and modify the data used by PIN and consequently ISR. Such illegal accesses may occur due to a program error, and could potentially be exploited by an attacker. For instance, an attacker could attempt to overwrite a function pointer or return address in PIN, so that control is diverted directly into the attacker's code in the application. Such a control transfer would circumvent ISR enabling the attacker to successfully execute his code. To defend against such attacks we need to protect PIN's memory from being written by the application.

When PIN loads and before the target application and its libraries gets loaded, we mark the entire address space as being "owned" by PIN, by asserting a flag in an array (*page-map*) that holds one byte for every addressable page. For instance, in a 32-bit Linux system, processes can typically access 3 out of the 4 GBytes that are directly addressable. For a page size of 4 KBytes, this corresponds to 786432 pages, so we allocate 768 KBytes to store the flags for the entire address space. As the target application gets loaded, and starts allocating additional memory, we update the flags for the application-owned pages. Memory protection is actually enforced by instrumenting all memory write operations performed by the application, and checking that the page being accessed is valid according to the page-map. If the application attempts to write to a page owned by PIN, the instrumentation causes a page-fault that will terminate it.

Memory protection further hardens the system against code-injection attacks, but incurs a substantial overhead. However, forcing an attacker to exploit a vulnerability in this fashion is already hardening the system considerably, as he would have to somehow discover one of the few memory locations that can be used to divert PIN's control flow. Alternatively, we can use address space layout randomization to decrease the probability of an attacker successfully guessing the location of PIN's control data.

Exceptions As we previously mentioned in Section 3, there are cases where certain external non-randomized instructions need to be executed in the context of the process, like in the case of signal trampolines. When a signal is delivered to a process, we scan the code being executed to identify *trampolines*, and execute them without applying the decoding function. In the case of a shared object like the VDSO, we assign its memory range a null key, which does not require it to be randomized. Since it is a read-only object, we can safely do so.

Multiple Instruction Sets Most executables in modern OSs are dynamically linked to one or more shared libraries. Shared libraries are preferred because they accommodate code reuse and minimize memory consumption, as their code can be concurrently mapped and used by multiple applications. As a result, mixing shared libraries with ISR has proved to be problematic in past work. Our implementation of ISR in software supports multiple instruction sets (*i.e.,* multiple randomization keys) for the same process, enabling us to support truly shared and randomized libraries.

We create a randomized copy of all libraries that are needed, and store them in a shadow folder (*e.g.,* "/usr/rand_lib"). Each library is encoded using a different key, and for extended randomization we can use a different key for each function within the library. To use these libraries, we modify the runtime environment, so that when an application is loaded, it first looks for shared libraries in the shadow folder. This way we can keep the original libraries in the usual system locations (*e.g.,* "/usr/lib" and "/lib" on Linux, and "c:\windows\system32" for Windows).

Protection from Unauthorized Binaries Implementing our extension to ISR in software-only is less attractive, mainly because of the performance overhead (discussed in Section 4.2). Because of this overhead, it will be probably applied only on selected applications, like network services. Nonetheless, if we desire to run all the processes in the system under ISR using PIN, we can modify the *init* process to launch all processes using PIN. This would cause all processes started later on (*e.g.,* via *exec()*) to also run under PIN and ISR.

4.2 Performance

Dynamic instrumentation tools usually incur significant slowdowns on target applications. While this is also true for PIN, we show that the overhead is not prohibitive. We conducted the measurements presented here on a DELL Precision T5500 workstation with a dual 4-core Xeon CPU and 24GB of RAM running Linux.

Figure 1 shows the mean execution time and standard deviation when running several commonly used Linux utilities. We draw the execution time for running *ls* on a directory with approximately 3400 files, and running *cp*, *cat*, and *bunzip2* with a 64MB file. We tested four execution scenarios: native execution, execution with PIN and no instrumentation (PIN's minimal overhead),

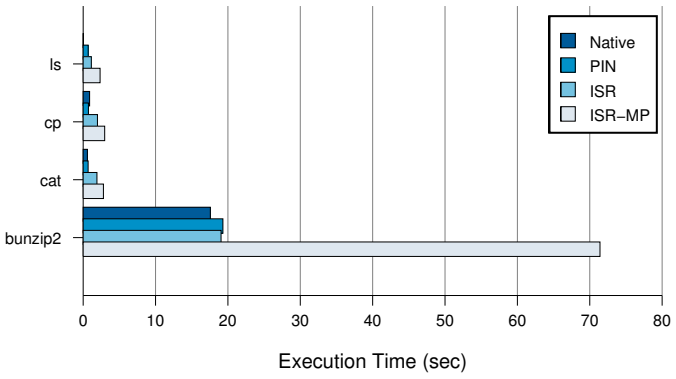

Fig. 1. Execution time of basic Linux utilities. The figure draws the mean execution time and standard deviation when running four commonly used Linux utilities.

our implementation of ISR without memory protection (MP), and lastly with MP enabled (ISR-MP). The figure shows that short-lived tasks suffer more, because the time needed to initialize PIN is relatively large when compared with the task's lifetime. In opposition, *when executing a longer-lived task, such as bunzip2, execution under ISR only takes about 10% more time to complete.*

For all four utilities, when employing memory protection to protect PIN's memory from interference, execution takes significantly longer, with bunzip2 being the worst case requiring *almost 4 times* more time to complete. That is because memory protection introduces additional instructions at runtime to check the validity of all memory write operations. Another interesting observation is that running bunzip2 under ISR is slightly faster from just using PIN. We attribute this to the various optimizations that PIN introduces when actual instrumentation is introduced.

We also evaluate our implementation using two of the most popular open-source servers: the *Apache* web server, and the *MySQL* database server. For Apache, we measure the effect that PIN and ISR have on the maximum throughput of a static web page, using Apache's own benchmarking tool *ab* over a dedicated 1Gb/s network link. To avoid high fluctuations in performance due to Apache forking extra processes to handle the incoming requests in the beginning of the experiment, we configured it to pre-fork all worker processes (pre-forking is a standard multi-processing Apache module), and left all other options to their default setting.

Figure 2 shows the mean throughput and standard deviation of Apache for the same four scenarios used in our first experiment. The graph shows that Apache's throughput is more limited by available network bandwidth than CPU power. Running the server over PIN has no effect on the attainable throughput, while applying ISR, even with memory protection enabled, does not affect server throughput either.

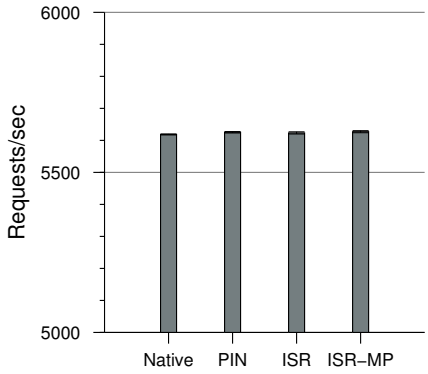

Fig. 2. Apache web server throughput. The figure draws the mean reqs/sec and standard deviation as measured by Apache's benchmark utility *ab*.

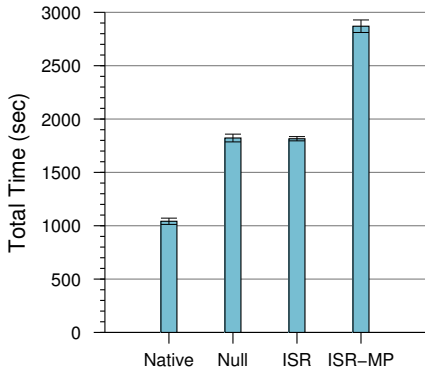

Fig. 3. The MySQL *test-insert* benchmark measures various SQL operations. The figure draws total execution time as reported by the benchmark utility.

Finally, we benchmarked a MySQL database server using its own *test-insert* benchmark, which creates a table, fills it with data, and selects the data. Figure 3 shows the time needed to complete this benchmark for the same four scenarios. PIN introduces a 75% overhead compared with native execution, while our ISR implementation incurs no observable slowdown. Unlike Apache, enabling memory protection for MySQL is 57.5% slower that just using ISR (175% from native). As with Apache, the benchmark was run at a remote client over an 1Gb/s network link to avoid interference with the server.

5 Perl Randomization

We showed that using ISR with binaries can protect us from code-injection attacks, and malicious binaries being executed without authorization. Most probably, attackers would adopt new attack vectors that would allow them to bypass ISR-enabled systems. For example, they may attempt to exploit input sanitization errors in interpreted scripts to inject and execute script code. Fortunately, the concept of ISR is particularly versatile, and can be applied to also protect such languages from command injection. We demonstrate its applicability by implementing ISR for the Perl language.

We randomize all of Perl's keywords, operators, and function calls, by appending a random 9-digit number ("tag") suffix to each of them. 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} };
}
```

Consequently, Perl code injected by an attacker will fail to execute, since the parser will not recognize a plain-text (not randomized) keyword, function, *etc.*

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 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. We automated the process of generating randomized code using Perltidy [28], 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.

This randomization scheme presents us with two problems. First, Perl's external modules that play the role of code libraries, and are frequently shared by many scripts and users. Second, the randomization key is provided in the command line, meaning that an attacker could drop his malicious randomized Perl script in the file system, and execute it.

To address these two issues, we define a system-wide key known only to the Perl interpreter. Using this scheme, the administrator can periodically randomize the system modules, without requiring any action from the user, while an attacker will have to successfully guess the "tag" used by the system to successfully run his malicious Perl script. For a 9- digit "tag" that would require at most $10^9 attempts$, but unlike before we can use an (almost) arbitrarily long "tag". Finally, although the size of the scripts increases considerably due to the randomization process, some preliminary measurements indicate that performance is unaffected.

Shell scripts can be randomized in a similar way to defend against shell injection attacks [29]. For instance,

```
#!/bin/sh
if987654 [ x$1 ==987654 x"" ]; then987654
  echo987654 "Must provide directory name."
  exit987654 1
fi987654
```

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 interpreted languages that could benefit from ISR include VBS, Python, and others.

6 SQL Randomization

SQL-injection attacks have serious security and privacy implications [30], specially because they require little effort on the behalf of the attacker. Most frequently, they are used against web applications that accept user input, and use it to compose SQL queries. When these applications do not properly sanitize user inputs, an attacker can carefully craft inputs to inject SQL statements that can potentially allow him to access or corrupt database (DB) data, modify DB structures, *etc.*

For example, consider a log-in 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 user-name, 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 first single quotes balance the opening quote after *username*, and the remainder of the attacker's input is evaluated as an SQL script by the DB. In this case, *or 1=1* will result in the query returning all the records in *mysql.user*, since the *where* clause evaluates to true. The double hyphen comments out the rest of the SQL query to ensure that no error occurs. If an application uses the above query to determine whether a user's credentials are valid, an attacker supplying the above input would bypass the security check, resulting in a successful log in.

SQL-injection attacks are frequently used to gain unauthorized access to web sites, and then extract sensitive information. For instance, an attacker could read a randomized binary from the file system to launch a brute-force attack against its key, and (after discovering it) launch a successful code-injection attack. *We extend ISR to the SQL language to protect against such information leaking attacks.* We randomize SQL's standard operators (including keywords, mathematical operators, and other invariant language tokens) by appending a random integer (like the 9-digit "tag" used for Perl randomization). All SQL injection attacks are then prevented, because the user input inserted into the "randomized" query is classified as a set of non-operators, resulting in an invalid expression.

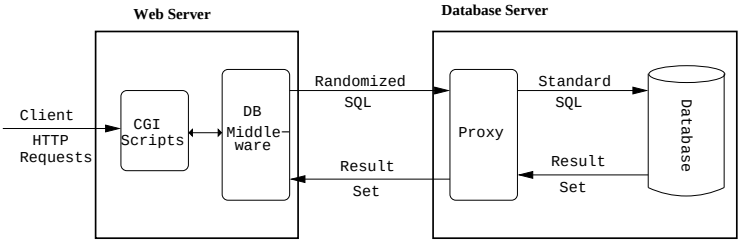

Fig. 4. SQLrand System Architecture

Essentially, we introduce a new set of keywords to SQL that will not be recognized by the DB's SQL interpreter. Unlike Perl randomization in Section 5, modifying the DB's interpreter to accept the new set of keywords is complicated. Furthermore, a modified DB engine would require that all applications using it conform to its new language. Although running a dedicated DB server for the web applications that we want to protect may be feasible, they would still be forced to all use the same random key.

Our design allows for a single DB server that can be used with multiple web applications employing multiple randomization keys, while at the same time it can be used by non-randomized applications. It consists of a proxy that sits between the client and database server as shown in Figure 4. By moving the de-randomization process outside the database management system (DBMS), we gain flexibility, simplicity, and security. Multiple proxies using different random keys can be listening for connections on behalf of the same database. Each proxy deciphers the randomized SQL query received, and then forwards it to the DB. It is also responsible for concealing DB errors which may reveal the key used by the application to the attacker. For example, the attacker could perform a simple SQL-injection to cause a parse error that would return an error message. This message could 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.

We assist the developer to randomize his SQL statements, by providing a tool that reads a SQL statements 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 example 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
```

The generated SQL query can be inserted into the developer's web application. The proxy receives the randomized SQL, translates and validates it, before forwarding it to the database. Note that the proxy performs simple syntactic validation, but is otherwise unaware of the semantics of the query itself.

6.1 SQLrand Implementation

We built a proof-of-concept proxy server that implements ISR for the SQL language. The proxy detects SQL-injection attacks, and rejects the malicious queries, so they never reach the DB server. It consists of two components, the de-randomization (or decoding) element, and the communication component that implements the communication protocol between the DB client and server. Our implementation focuses on CGI scripts being the query generators and DB clients, but our approach can be also applied to other environments like Java, and the Java database access framework (JDBC).

The decoding component is essentially an SQL parser that can "understand" the randomized SQL language. To create the parser, we utilized two popular tools frequently used in the development of compilers: flex and yacc. First, we used flex and regular expressions to match SQL keywords followed by zero or more digits, so we can capture the encoded SQL queries. (Technically, it did not require a key; practically, it needs one.) The lexical analyzer uses these expressions to strip the random extension, and return the keyword for grammar processing and query reassembly by yacc. Tokens that did not match are labeled as identifiers. During parsing, any syntax error indicates that either the query was not properly randomized with the pre-selected key, or that an SQL-injection attack is taking place. In both cases, when an error occurs the parser returns NULL, while successfully parsing the query returns the de-randomized SQL string.

For the database server, we used MySQL, one of the most popular open-source DB systems. Our communication component implements MySQL's protocol between the proxy and the client, as well as between the proxy and the server. To communicate with the server, we used MySQL's C client library, which was sufficient. On the other hand, a server-side library implementing MySQL's protocol was not available. Therefore, we resorted to manually analyzing the MySQL protocol to obtain a rough sketch of the basics of the protocol: querying, error reporting, and disconnecting. The query message, as the name implies, carries the actual requests to the server. The disconnect message is necessary in cases where the client abruptly disconnects from the proxy, or sends the proxy an invalid query. In both cases, the proxy is responsible for disconnecting from

the database by issuing the disconnect command on behalf of the client. Finally, the error message is sent to the client when an query generates a syntax error, indicating a possible injection attack.

Configuring a client application to use the proxy is straightforward. Assuming that the proxy is running on the same host as the server, it is adequate to modify the client to use the port of the proxy instead of the server's. After receiving a connection, the proxy establishes a connection with the database, where it forwards the messages it receives from the client. Messages that contain queries, and are successfully parsed by the proxy, are forwarded to the server. If parsing of a query fails, the proxy returns a generic syntax error to the client (so as not to reveal the randomization key), and disconnects from the server.

6.2 Limitations

Stored procedures Stored procedures are SQL statements that are stored in the DB itself, and are invoked by the client as a function (*e.g., select new_employee(id, name, department)*). They are also susceptible to SQL-injection attacks, but cannot be protected using the current SQLrand design. In particular, it is impossible to de-randomize them without changing the SQL parsing logic in the database. Additionally, using a variable randomization key may also be problematic, depending on the DB's implementation. A potential solution could involve storing the queries in an external data source (*e.g.,* an XML file) that the application reads during execution. These queries can be randomized during runtime under a different key.

Problematic library calls The client library may define some API methods, which use fixed queries. For instance, MySQL's *mysql_list_dbs()* call issues the query string "SHOW databases LIKE <wild-card-input>", which is hard coded. We could workaround this issue without modifying the client library, by manually constructing the query string with the proper randomized key, and executing it using the *mysql_query()* method. Moreover, pre-compiled binary SQL statements cannot be currently processed by the proxy, therefore *mysql_real_query()* must be avoided.

6.3 Evaluation

We created a small database with tables containing various numbers of records, ranging from twenty to a little more than a thousand, to evaluate the effectiveness and performance of SQLrand.

First, we wrote a sample CGI application that suffers from an SQL-injection vulnerability, which allows an attacker to inject SQL into the *where* clause of the query. An attacker could easily exploit this fault to retrieve all the records in the table, even though he should not be allowed to. When using the SQLrand proxy, the SQL-injection was identified and an error message was returned instead of the table data.

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

Table 2. Proxy Overhead (in microseconds)

Then, we tested SQLrand with existing software like the phpBB v2.0.5 web bulletin board system (BBS). The script *viewtopic.php* of the BBS is vulnerable to an SQL-injection attack that can reveal critical information like user passwords. We first performed the SQL-injection attack without using randomization, and after ensuring it succeeded, we randomized the SQL queries in the script, and configured the BBS to use the SQLrand proxy. As expected, when we relaunched the attacked, the proxy recognized it, and dropped the query without forwarding it to the DB. While the phpBB application did not succumb to the SQL-injection attack, we observed that it displayed the SQL query when zero records are returned, revealing the encoding being used. So, while ISR stops SQL-injection attacks, it can be of little benefit, when bad coding practices result in critical information about the application being divulged to potential attackers.

We also tested another content management system (CMS) that is prone to SQL injection attacks. The PHP-Nuke CMS, depends on PHP's *magic_quotes_gpc* option being enabled, or otherwise several modules are open to attack. Interestingly, even with this option set, injections using unchecked numeric fields are still possible. For example, PHP-Nuke uses the numeric variable *lid*, which is passed through the URL when downloading content. An attacker can perform SQL-injection using this variable, to retrieve all user passwords (*e.g.,* by appending *select pass from users_table* to an invalid *lid* value). However, when using SQLrand this attack was averted.

Next, we measured the overhead imposed by SQLrand. We designed an experiment to measure the additional processing time required, when multiple users (*i.e.,* 10, 25, and 50 respectively) perform queries concurrently. The users executed, in a round-robin fashion, a set of five queries over 100 trials. The average length of the queries was 639 bytes, and the random key length was 32 bytes. For this experiment, the DB, proxy, and client were all running on different hosts running RedHat Linux, within the same network. Table 2 lists the results of our experiments. We see that the overhead ranges from 183 to 316 microseconds, and in the worst-case scenario the proxy adds approximately 6.5 milliseconds 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, we believe that the additional delay introduced by the proxy is acceptable for the majority of applications.

7 Security Considerations

Performing code injection in a few vulnerable applications running under ISR, caused the targets to terminate with a segmentation violation or illegal opcode exception. Barrantes *et al.* [19] performed a study on the faults exhibited by a compromised process running under ISR, and show that such a process executes 5 or less x86 instructions before causing a fatal exception. The instructions that get actually executed are essentially random bytes produced by the de-randomization of the attacker's injected code.

The rest of this section discusses the methods that might be employed by an attacker to bypass instruction-set randomization.

Key Guessing Attacks The most obvious way to attack ISR is by attempting to guess the key, or one of the keys used by a process. As we re-randomize a process with a new key every time it crashes, such brute-force attacks become purely a game of chance. For instance, when using a 16-bit key the attacker has an $1/2^{16}$ probability to guess the key correctly in the first attempt. An incorrect attempt to guess the key, will cause the process to fail and re-randomize. Even if the attacker manages to attack part of the key (the 8 bits corresponding to a single-byte instruction), as shown in [20], he would still have only an $1/2^8$ probability to guess the key correctly. Furthermore, Cox *et al.* [31] showed that a quorum architecture, where each replica is randomized differently (with a different key), would defeat all key guessing attacks.

Some server processes use *fork()* to create a copy of the parent process every time a new request is received to handle that request. An attacker can then attempt multiple guesses against the same key. In such cases, the *fork()* system call can be itself modified, so that if the process is employing instruction-set randomization, the text segment is actually copied (rather than just copying the page table entries of the parent) and re-randomized. 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 [32–35]. Alternatively, the parent process can be restarted every time a child process crashes. This would ensure that the parent process itself and new children processes will use a new key, while processes already serving clients will keep operating normally.

Known Ciphertext Attacks Since binary code is highly structured, an attacker with access to the randomized code of a process can easily determine the randomization key, and create valid attack payloads. An attacker may easily gain access to randomized code, if he already has local access to the system (*e.g.,* through the */proc* interface on Linux). Since local access is already available to him, he would then target processes running with higher privileges than his own (a privilege escalation attack). We could mitigate this problem by using a stronger encryption algorithm such as AES or bit transposition for the randomization, possibly taking a performance hit. As we mentioned earlier, our

technique is primarily focused on deterring remote attackers from gaining local access through code injection. For this task XOR encryption remains sufficient.

However, even for remote attackers, it is imperative that the system does not expose information that could be used by the attacker to increase his chances to brute-force attack ISR. For instance, SQL-injection could be used by an attacker to gain partial access on the system, and obtain a randomized binary. He can then launch a brute-force attack against its key, and later perform a successful code-injection attack. In Section 6, we discussed how we can apply ISR on SQL to defeat such attacks.

Attacks Using Interpreters With the application of our approach, attackers will no longer be able to execute binary code on the system. As a result, they may resort to attacking applications written in an interpreted or scripting language, or to simply drop and execute such a script on the targeted system. In Section 5, we described how ISR can be applied on an interpreted language like Perl, and argued that it can be applied with little effort on other languages as well. As many such interpreters are frequently present on a system concurrently, the process of identifying and securing all of them using ISR becomes problematic. For instance, in Linux we find multiple shell binaries that implement various scripting languages (*e.g.,* bash, tcsh, ksh, *etc.*), and interpreted programming environments like Perl, Python, Tcl, *etc.* An attacker would need only one of them to be running without ISR to subvert our defenses.

We can prevent such attacks by identifying the interpreters present on a system, and requiring that they are ISR-enabled, or disallowing their execution. Frequently, scripts written in such languages begin by specifying a "magic number" (*i.e., '#!'*) followed by the location and name of the interpreter required for execution (*e.g., #!/usr/bin/perl*). When such scripts execute, we can modify the kernel to look for this string, and check if the executed interpreter uses randomization, or whether it is allowed to execute. Unfortunately, scripts can be also run by directly invoking the interpreter binary, even if they do not contain such a magic number (*e.g., /usr/bin/perl myscript.pl*).

Alternatively, we can scan the file system for files beginning with '#!' to statically identify existing interpreters. While not all script files contain this magic number, it is more likely that at least one script for each installed interpreter will exist that contains it. For example, even if all Perl scripts observed running do not begin with the magic number, there will be other Perl scripts in the file system (*e.g.,* Perl modules) that begin with it. More elaborate solutions, could even employ static and dynamic analysis to detect interpreter binaries based on their features and behavior (*e.g.,* detect their lexical analyzer).

8 Related Work

Instruction-set randomization was initially proposed as a general approach against code-injection attacks by Gaurav *et al.* [17]. They propose a low-overhead implementation of ISR in hardware, and evaluate it using the Bochs x86 emu-

lator. They also demonstrate the applicability of the approach on interpreted languages such as Perl, and later SQL [36]. Concurrently, Barrantes *et al.* [18] proposed a similar randomization technique for binaries (RISE), which builds on the Valgrind x86 emulator. RISE provides limited support for shared libraries by creating randomized copies of the libraries for each process. As such, the libraries are not actually shared, and consume additional memory each time they are loaded. Furthermore, Valgrind incurs a minimum performance overhead of 400% [37], which makes its use impractical.

Hu *et al.* [26] implemented ISR using a virtual execution environment based on a dynamic binary translation framework named STRATA. Their implementation uses AES encryption with a 128-bit key, which requires that code segments are aligned at 128-bit blocks. Unlike our implementation over PIN, they do not support self-modifying code, and they produce randomized binaries that are significantly larger from the originals (*e.g.,* the randomized version of Apache was 77% larger than the original). Also, to the best of our knowledge previous work on ISR does not address the implications introduced by signal trampolines and VDSO, nor does it investigate the costs involved with protecting the execution environment from the hosted process (STRATA protects only a part of its data).

Address obfuscation is another approach based on randomizing the execution environment (*i.e.,* the locations of code and data) to harden software against attacks [38, 39]. It can be performed at runtime by randomizing the layout of a process (ASLR) including the stack, heap, dynamically linked libraries, static data, and the process's base address. Additionally, it can be performed at compile time to also randomize the location of program routines and variables. Shacham *et al.* [40] show that ASLR may not be very effective on 32-bit systems, as they do not allow for sufficient entropy. In contrast, Bhatkar *et al.* [41] argue that it is possible to introduce enough entropy for ASLR to be effective. Meanwhile, attackers have successfully exploited ASLR enabled systems by predicting process layout, exploiting applications to expose layout information [42], or using techniques like heap spraying [43].

Hardware extensions such as the NoExecute (NX) bit in modern processors [39, 44] can stop code-injection attacks all together without impacting performance. This is accomplished by disallowing the execution of code from memory pages that are marked with the NX bit. Unfortunately, its effectiveness is dependent on its proper use by software. For instance, many applications like browsers do not set it on all data segments. This can be due to backward compatibility constraints (*e.g.,* systems using signal trampolines), or even just bad developing practice. More importantly, NX does not protect from unauthorized code execution.

PointGuard [45] uses encryption to protect pointers from buffer overflows. It encrypts pointers in memory, and decrypts them only when they are loaded to a register. It is implemented as a compiler extension, so it requires that source code is available for recompilation. Also, while it is able to deter buffer overflow attacks, it can be defeated by format string attacks that frequently employ code-injection later on. Other solutions implemented as compiler extensions in-

clude Stackguard [46] and ProPolice [47]. They operate by introducing special secret values in the stack to identify and prevent stack overflow attacks, but can be subverted [48]. Write integrity testing [49] uses static analysis and "guard" values between variables to prevent memory corruption errors, but static analysis alone cannot correctly classify all program writes. CCured [50] is a source code transformation system that adds type safety to C programs, but it incurs a significant performance overhead and is unable to statically handle some data types. Generally, solutions that require recompilation of software are less practical, as source code or parts of it (*e.g.,* third-party libraries) are not always available.

Dynamic binary instrumentation is used by many other solutions to retrofit unmodified binaries with defenses against remote attacks. For instance, dynamic taint analysis (DTA) is used by many projects [32,51–53], and is a able to detect control hijacking and code-injection attacks, but incurs large slowdowns (*e.g.,* frequently 20x or more). Due to their large overhead, dynamic solutions are mostly used for the analysis of attacks and malware [54], and in honeypots [55].

9 Conclusions

Instruction-set randomization (ISR) is a powerful scheme that can protect binaries from code-injection attacks regardless of how code is injected within a process, by presenting a moving target to an attacker that is attempting to inject malicious code to the system. The original ISR scheme, despite its versatility, could not protect against the most rudimentary type of attack, such as the execution of a malicious and unauthorized binaries. We propose the whole-sale adoption of ISR across all layers of the software stack to protect against such attacks. By pre-randomizing all of a system's binaries with different and most importantly secret keys, the execution of binaries placed on the target by an attacker or worm like Stuxnet will fail to execute in the ISR-enabled environment. At the same time, new binaries can be installed using an installation script that requires user authorization, and that automatically randomizes the installed binary. We also describe in detail how ISR can be implemented in hardware, as well as entirely in software. Finally, we show how ISR can be applied on interpreted languages like Perl and SQL, which may be targeted next by attackers that wish to circumvent ISR. In both cases, we demonstrate that ISR is extremely versatile and can be applied on both Perl and SQL with success and low overhead.

ISR does not address the core issue of software vulnerabilities, which derive from programming errors, and bad coding practices. Nevertheless, given the apparent resistance to the wide adoption of safe languages, and the recent rise of extremely elaborate worms like Conficker and Stuxnet, we believe that ISR can play an important role in hardening systems.

Acknowledgements

This work was supported by the NSF through Grant CNS-09-14845, and by the Air Force through Contracts AFOSR-FA9550-07-1-0527 and AFRL-FA8650-10-C-7024. Any opinions, findings, conclusions or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, the Air Force, or the NSF.

References

1. Wagner, D., Foster, J.S., Brewer, E.A., Aiken, A.: A first step towards automated detection of buffer overrun vulnerabilities. In: Proceedings of the Symposium on Network and Distributed System Security (NDSS). (2000) 3–17
2. Spafford, E.H.: The Internet worm program: An analysis. Technical Report CSD-TR-823, Purdue University (1988)
3. CERT: Advisory CA-2001-19: "Code Red" worm exploiting buffer overflow in IIS indexing service DLL. `http://www.cert.org/advisories/CA-2001-19.html` (2001)
4. CERT: Advisory CA-2003-04: MS-SQL Server Worm. `http://www.cert.org/advisories/CA-2003-04.html` (2003)
5. Moore, D., Shanning, C., Claffy, K.: Code-Red: a case study on the spread and victims of an Internet worm. In: Proceedings of the 2^{nd} Internet Measurement Workshop (IMW). (2002) 273–284
6. Zou, C.C., Gong, W., Towsley, D.: Code Red worm propagation modeling and analysis. In: Proceedings of the 9^{th} ACM Conference on Computer and Communications Security (CCS). (2002) 138–147
7. Porras, P., Saidi, H., Yegneswaran, V.: Conficker C analysis. Technical report, SRI International (2009)
8. Falliere, N., Murchu, L.O., Chien, E.: W32.Stuxnet Dossier version 1.2. White paper (2010)
9. Adobe: Security advisory for flash player, adobe reader and acrobat. `http://www.adobe.com/support/security/advisories/apsa10-01.html` (2010)
10. Symantec: Analysis of a zero-day exploit for adobe flash and reader. Symantec Threat Research (2010)
11. Pincus, J., Baker, B.: Beyond stack smashing: Recent advances in exploiting buffer overflows. IEEE Security & Privacy Magazine **2** (2004) 20–27
12. Aleph One: Smashing the stack for fun and profit. Phrack **7** (1996)
13. M. Conover and w00w00 Security Team: w00w00 on heap overflows. `http://www.w00w00.org/files/articles/heaptut.txt` (2010)
14. Enumeration, C.W.: CWE-416: use after free. `http://cwe.mitre.org/data/definitions/416.html` (2010)
15. PCWorld: Dangling pointers could be dangerous. `http://www.pcworld.com/article/134982/dangling_pointers_could_be_dangerous.html` (2007)
16. Shankar, U., Talwar, K., Foster, J.S., Wagner, D.: Detecting format string vulnerabilities with type qualifiers. In: Proceedings of the 10^{th} USENIX Security Symposium. (2001) 201–216
17. Kc, G.S., Keromytis, A.D., Prevelakis, V.: Countering code-injection attacks with instruction-set randomization. In: Proceedings of the 10^{th} ACM Conference on Computer and Communications Security (CCS). (2003)

18. Barrantes, E.G., Ackley, D.H., Forrest, S., Palmer, T.S., Stefanovic, D., Zovi, D.D.: Randomized instruction set emulation to disrupt binary code injection attacks. In: Proceedings of the ACM Conference on Computer and Communications Security. (2003) 281–289
19. Barrantes, E.G., Ackley, D.H., Forrest, S., Stefanović, D.: Randomized instruction set emulation. ACM Transactions on Information System Security **8** (2005) 3–40
20. Sovarel, A.N., Evans, D., Paul, N.: Where's the FEEB? the effectiveness of instruction set randomization. In: Proceedings of the 14^{th} USENIX Security Symposium. (2005) 145–160
21. Microsoft: Microsoft Portable Executable and Common Object File Format Specification. `http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx` (2010)
22. Raghuram, S., Chakrabarti, C.: A programmable processor for cryptography. In: Proceedings of the 2000 IEEE International Symposium on Circuits and Systems (ISCAS). Volume 5. (2000) 685–688
23. Rogers, B., Solihin, Y., Prvulovic, M.: Memory Predecryption: Hiding the Latency Overhead of Memory Encryption. In: Proceedings of the Workshop on Architectural Support for Security and Anti-virus (WASSA). (2004) 22–28
24. The Bochs Project: The cross platform IA-32 emulator. `http://bochs.sourceforge.net/` (2010)
25. Prevelakis, V., Keromytis, A.D.: Drop-in Security for Distributed and Portable Computing Elements. Internet Research: Electronic Networking, Applications and Policy **13** (2003)
26. Hu, W., Hiser, J., Williams, D., Filipi, A., Davidson, J.W., Evans, D., Knight, J.C., Nguyen-Tuong, A., Rowanhill, J.: Secure and practical defense against code-injection attacks using software dynamic translation. In: Proceedings of the 2^{nd} International Conference on Virtual Execution Environments (VEE). (2006) 2–12
27. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proceedings of Programming Language Design and Implementation (PLDI). (2005) 190–200
28. Hancock, S.: The Perltidy Home Page. `http://perltidy.sourceforge.net/` (2009)
29. CERT: Vulnerability Note VU#496064. `http://www.kb.cert.org/vuls/id/496064` (2002)
30. CERT: Vulnerability Note VU#282403. `http://www.kb.cert.org/vuls/id/282403` (2002)
31. Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., Hiser, J.: N-Variant Systems: A Secretless Framework for Security through Diversity. In: Proceedings of the 15^{th} USENIX Security Symposium. (2005) 105–120
32. Costa, M., Crowcroft, J., Castro, M., Rowstron, A.: Vigilante: End-to-end containment of internet worms. In: Proceedings of the ACM Symposium on Systems and Operating Systems Principles (SOSP). (2005)
33. Xu, J., Ning, P., Kil, C., Zhai, Y., Bookholt, C.: Automatic Diagnosis and Response to Memory Corruption Vulnerabilities. In: Proceedings of the 12^{th} ACM Conference on Computer and Communications Security (CCS). (2005) 222–234
34. Locasto, M., Wang, K., Keromytis, A., Stolfo, S.: FLIPS: Hybrid Adaptive Intrusion Prevention. In: Proceedings of the Symposium on Recent Advances in Intrusion Detection. (2005) 82–101

35. Liang, Z., Sekar, R.: Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers. In: Proceedings of the 12^{th} ACM Conference on Computer and Communications Security (CCS). (2005) 213–222
36. Boyd, S.W., Kc, G.S., Locasto, M.E., Keromytis, A.D., Prevelakis, V.: On the general applicability of instruction-set randomization. IEEE Transactions on Dependable and Secure Computing **99** (2008)
37. Developers, V.: Valgrind user manual – callgrind. `http://valgrind.org/docs/manual/cl-manual.html` (2010)
38. Bhatkar, S., DuVarney, D.C., Sekar, R.: Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In: Proceedings of the 12^{th} USENIX Security Symposium. (2003) 105–120
39. The PaX Team: Homepage of The Pax Team. `http://pax.grsecurity.net/` (2010)
40. Shacham, H., Page, M., Pfaff, B., Goh, E., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: Proceedings of the 11^{th} ACM Conference on Computer and Communications Security (CCS). (2004) 298–307
41. Bhatkar, S., Sekar, R., DuVarney, D.C.: Efficient techniques for comprehensive protection from memory error exploits. In: Proceedings of the 14^{th} USENIX Security Symposium. (2005) 255–270
42. Durden, T.: Bypassing PaX ASLR protection. Phrack **0x0b** (2002)
43. DarkReading: Heap spraying: Attackers' latest weapon of choice. `http://www.darkreading.com/security/vulnerabilities/showArticle.jhtml?articleID=221901428` (2009)
44. Hardware, E.: CPU-based security: The NX bit. `http://hardware.earthweb.com/chips/article.php/3358421` (2004)
45. Cowan, C., Beattie, S., Johansen, J., Wagle, P.: PointGuard: Protecting pointers from buffer overflow vulnerabilities. In: Proceedings of the 12^{th} USENIX Security Symposium. (2003) 91–104
46. Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 7^{th} USENIX Security Symposium. (1998)
47. Etoh, J.: GCC extension for protecting applications from stack-smashing attacks. `http://www.trl.ibm.com/projects/security/ssp/` (2000)
48. Bulba, Kil3r: Bypassing StackGuard and StackShield. Phrack **5** (2000)
49. Akritidis, P., Cadar, C., Raiciu, C., Costa, M., Castro, M.: Preventing memory error exploits with WIT. In: Proceedings of the 2008 IEEE Symposium on Security and Privacy. (2008) 263–277
50. Necula, G.C., Condit, J., Harren, M., McPeak, S., Weimer, W.: CCured: type-safe retrofitting of legacy software. ACM Trans. Program. Lang. Syst. **27** (2005) 477–526
51. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proceedings of the 12^{th} Annual Symposium on Network and Distributed System Security (NDSS). (2005)
52. Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19** (1976) 236–243
53. Ho, A., Fetterman, M., Clark, C., Warfield, A., Hand, S.: Practical taint-based protection using demand emulation. In: Proceedings of the 1^{st} ACM EuroSys Conference. (2006) 29–41

54. Bayer, U., Kruegel, C., Kirda, E.: TTAnalyze: A tool for analyzing malware. In: Proceedings of the 15^{th} European Institute for Computer Antivirus Research (EICAR) Annual Conference. (2006)
55. Portokalidis, G., Slowinska, A., Bos, H.: Argos: an emulator for fingerprinting zero-day attacks. In: Proceedings of the 1^{st} ACM EuroSys Conference. (2006)