Fast and Practical Instruction-Set Randomization for Commodity Systems

Georgios Portokalidis and Angelos D. Keromytis
Network Security Lab
Department of Computer Science
Columbia University, New York, NY, USA
{porto, angelos}@cs.columbia.edu

ABSTRACT

Instruction-set randomization (ISR) is a technique based on randomizing the "language" understood by a system to protect it from code-injection attacks. Such attacks were used by many computer worms in the past, but still pose a threat as it was confirmed by the recent Conficker worm outbreak, and the latest exploits targeting some of Adobe's most popular products. This paper presents a fast and practical implementation of ISR that can be applied on currently deployed software. Our solution builds on a binary instrumentation tool to provide an ISR-enabled execution environment entirely in software. Applications are randomized using a simple XOR function and a 16-bit key that is randomly generated every time an application is launched. Shared libraries can be also randomized using separate keys, and their randomized versions can be used by all applications running under ISR. Moreover, we introduce a key management system to keep track of the keys used in the system. *To the best of our knowledge we are the first to apply ISR on truly shared libraries.*

Finally, we evaluate our implementation using real applications including the Apache web server, and the MySQL database server. For the first, we show that our implementation has negligible overhead (less than 1%) for static HTML loads, while the overhead when running MySQL can be as low as 75%. We see that our system can be used with little cost with I/O intensive network applications, while it can also be a good candidate for deployment with CPU intensive applications, in scenarios where security outweighs performance.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Invasive software

General Terms

Security, Reliability, Performance

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

Keywords

Code-injection, randomization, security, performance

1. INTRODUCTION

Instruction-set randomization [25, 4] is a technique based on randomizing a system's language (*i.e.,* its instruction set) to prevent code-injection attacks. Such attacks occur when the attacker is able to execute arbitrary code remotely, or locally but as a different user (*e.g.,* a user with elevated privileges like the administrator). They usually follow the exploitation of buffer overflows [35, 3, 27] and other memory corruption vulnerabilities, like dangling pointers [20, 34] and format string attacks [39], that allow attackers to redirect execution to the injected code.

In the past, code-injection attacks (CI) accounted for almost half of the advisories released by CERT [43], and were used by many computer worms to infect new hosts [41, 11, 12, 29, 44]. More recently, they have been overshadowed by cross-site scripting and SQL-injection attacks, but the recent Conficker worm outbreak [36], and the multiple vulnerabilities discovered in Adobe's popular software [1, 42] came as a reminder that CI attacks still pose a significant threat to a large number of systems.

ISR is a general approach that defeats *all types* of remote code-injection regardless of the way it was injected into a process. It operates by randomizing the instructions that the underlying system "understands", so that "foreign" code such as the code injected during an attack will fail to execute. It was initially proposed as a modification to the processor to ensure low performance overheads, but unfortunately this proposal has had little allure with hardware vendors. Instead, software implementations of ISR on x86 emulators have been created, mainly to demonstrate the effectiveness of the approach, as they incur large runtime overheads [25, 4]. Software only implementations of ISR using dynamic binary translation have been also proposed [24], 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.

This paper describes a fast and practical software implementation of ISR for commodity systems. Our implementation is based on Intel's dynamic instrumentation tool called PIN [26], which provides the runtime environment. Application code is randomized using the XOR function and a 16-bit key, which is randomly generated every time the ap-

plication is launched to make it resistant against key guessing attacks [40].

Multiple keys can be used to randomize different parts of the application. For instance, every shared library used by the system can be randomized using a different key, creating a randomized copy of each library. While additional disk space will be required to store the randomized versions, during runtime all binaries running under ISR will be using the same randomized copy. Also, original (native) code can be combined with randomized code. The keys used to encode the various libraries are managed using SQLite [32], a self-contained and serverless database engine. Libraries can be randomized once and reused by multiple applications, while frequently re-randomizing them also protects them against key guessing attempts. Finally, we assume (as does past work) that the attacker does not have access to the randomized code (*i.e.,* it is a remote attacker), so a known ciphertext attack against the key is not possible.

The *main contributions* of this paper can be summarized in the following:

- We implemented instruction-set randomization for commodity systems using Intel's PIN framework (our implementation of ISR is freely available from `https://sourceforge.net/projects/isrupin/`)
- Our implementation operates on currently deployed binaries, as it does not require recompilation, or changes to the underlying system (*i.e.,* the operating system and hardware)
- Our system supports dynamically linked executables, as well as dynamically loaded libraries. We also introduce a key management scheme for storing and keeping track of the keys used to randomize shared libraries and applications. To the best of our knowledge we are the first to apply ISR on shared libraries
- Executables are re-randomized every time they are launched, and shared libraries are re-randomized at custom intervals to protect the key from guessing attacks such as [40]

The overhead of our implementation can be as low as 10% compared with native execution. We are able to run popular servers such as the Apache web server, and the MySQL database server, and show that running Apache using ISR has negligible effect on throughput for static HTML loads, while the overhead for running MySQL is 75%. We also evaluate the cost of completely isolating the framework's data from the application. This memory protection (MP) requires more invasive instrumentation of the target application, and it has not been investigated by previous work on software-based ISR, since it incurs significant overhead. We show that adding MP over ISR does not reduce Apache's throughput, while it imposes an extra 57% overhead when running MySQL.

The rest of this paper is organized as follows: Section 2 offers a brief description of ISR. Our implementation is discussed in Section 3. We evaluate the performance of our framework in Section 4. Related work is examined in Section 5. Finally, conclusions are in Section 6.

2. INSTRUCTION-SET RANDOMIZATION

Instruction-set randomization as a mean to thwart code-injection attacks has been presented in detail in previous work [25, 4]. In this section we will only briefly describe the technique, mainly focusing on its application on binaries.

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. In other words, the attacker needs to be able to "talk" to the target system in its own "language". For binary programs, 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. We should note that in this example it is possible to compose (somewhat limited) machine code able to run without errors on both ARM and x86.

ISR builds on this observation to block attackers from executing code injected in vulnerable processes. An execution environment employing a randomly generated instruction set is used to run processes, causing injected code to fail. While exploitation attempts will still cause a DoS by crashing the targeted application, attackers are not able to perform any useful action such as installing malware or rootkits. The strength of the technique lies in the difficulty of guessing the instruction set used by a process. Of course, if an attacker has access to the randomized binary, he can launch an attack against the applied transformation to attempt to learn the new instruction set, something that requires local access to the target host. This work (and ISR in general) is primarily focused on protecting against remote attacks on network services (*e.g.,* http, dns, ssh, *etc.*), where the attacker does not have access to the target system or the randomized binaries. Consequently, attackers cannot launch attacks against the key that require access to the ciphertext.

However, remote attackers can still attempt to guess the key used to randomize the instruction set [40]. Such guessing attacks will cause the application to crash and restart for each failed attempt to correctly guess the key. We can mitigate such attacks by either using a more complicated encoding algorithm (*e.g.,* bit transposition, AES, *etc.*) and a larger key to increase the complexity of the attack, or by frequently re-encoding the binary using a new key every time it is executed as we discuss below. The reader can refer to our earlier work on ISR [25] for additional discussion on randomization using larger keys.

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 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 2^{32} combinations in the worst case 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. On the other hand, simply increasing the length of the key used with XOR will not improve security, since the key can be attacked in a piece-meal fashion (*i.e.,* by guessing the first 32 bits of the key that correspond to a single instruction). *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 the key in four or two sub-keys of 8 and 16 bits respectively. Thus, it is possible that an attacker attempts to guess each of the sub-keys independently, as shown by Sovarel *et al.* [40].

The deficiencies of XOR randomization on architectures like the x86 can be overcome by using other ciphers for randomizing instructions. For instance, bit transposition of larger blocks (*e.g.,* 160 bits) would greatly increase the work factor for an attacker, and cannot be attacked in a piece-meal fashion. Hu et al [24] propose the use of AES encryption on blocks of 128 bits to ensure that an attacker cannot break the randomization. In both cases larger blocks of data need to be accessible at runtime, and more processing is required to decode the instructions. We have taken a different approach to protect the keys. First, we employ multiple keys for the encoding of an application (*i.e.,* a different key for each shared library). Second, we randomize an application every time it is launched with a new random key, and third we frequently re-randomize shared libraries.

Finally, we note that the security of the approach depends on the fact that injected code will raise an exception (*e.g.,* by accessing an illegal address or using an invalid opcode), after it has been de-randomized by the execution environment. 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. This number is statistically insignificant [5].

2.2 ISR Runtime

A randomized process requires the appropriate execution environment to de-randomize its instructions before they are executed. Previous work on ISR has demonstrated that it is possible to implement such an environment both in hardware and software. In both cases, the environment needs access to the key used during the randomization. The key can be stored within the executable, or in a database. Storing it within the application is compact and removes the need for external storage (*i.e.,* a DB), but could expose the key if the application leaks information.

Additionally, programs frequently make use of libraries, which may or may not be randomized. ISR needs to be able to detect when execution switches from a randomized piece of code to a plain one, and vice-versa. Detecting such context switches can be complex (specially in hardware), and in fact previous work has only handled statically linked executables. We will show in Section 3 that *our implementation is able to handle dynamically linked applications by supporting multiple instruction sets per process (i.e., instructions randomized with different keys).*

3. IMPLEMENTATION

We implemented ISR in software on 32-bit Linux for dynamically and statically linked ELF executables and libraries. This section describes the components of our implementation. It should be noted that while the current implementation works on Linux, it can be easily ported to other platforms also supported by the runtime.

3.1 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. Also, the ELF format completely separates code and data, including control data such as the procedure linkage table (PLT), making it an ideal format for applying binary randomization.

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 a 16-bit key. We also extended `objcopy` to randomize shared libraries in ELF format. Randomizing using XOR does not require that the target binary is aligned, so it does not increase its size or modify its layout.

While our current implementation is currently able to randomize only 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 [28].

3.2 Shared Libraries

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 proven to be problematic in past work. Our implementation of ISR supports multiple instruction sets (*i.e.,* multiple randomization keys) for the same process, enabling us to use plain shared libraries with a randomized executable. Furthermore, it enables us to randomize each library with its own key, and share it amongst all processes running under ISR like an ordinary shared library.

We create a randomized copy of all libraries that are needed, and store them in a shadow folder (*e.g.,* "/usr/rand_lib"). For stronger security, each library is encoded using a different key, while we can also periodically re-randomize all the libraries using new keys. When an application is loaded in the runtime environment, we modify its environment so it first looks for shared libraries in a shadow folder. If a randomized version of a library is not found, it proceeds to look for a plain version in the usual system locations (*e.g.,* "/usr/lib" and "/lib" on Linux, and "c:\windows\system32"

for Windows). Of course, a process can be forced to only use randomized code if that is required. Moreover, multiple shadow folders can be used concurrently. For instance, if a process crashes (*e.g.,* a crash could be triggered by a failed exploitation attempt), we may re-encode all shared libraries to thwart key guessing attacks.

3.3 Key Management

Supporting multiple instruction sets for every process notably increases the number of keys that are active in the system at any given time. Thus, key management becomes an important aspect of the system, and specially because shared libraries can be randomized with their own key, and multiple versions of the libraries may co-exist in the system. Previous work proposed to store keys within the ELF files, which removes the need for separate storage for the keys. While this approach is robust, it leaves keys vulnerable to exposure if an application leaks data because of a bug or an error. In the past information leakage has been exploited to bypass address space layout randomization (ASLR) [19]. Additionally, storing the key within the executable might not be feasible when using binary formats other than ELF.

Instead, we store the keys for executables and libraries in a database, using the *sqlite* database system. *Sqlite* is a software library that implements a self-contained, serverless SQL database engine. The entire database is stored in a single file, and it is accessed directly by our tool (using the *sqlite* library) without the need to run additional processes. The keys are indexed using the library's full path, and the operation of retrieving a key from the DB is fast. As it is an operation that it is only performed when an application is launched or a dynamic library is loaded, its performance is not critical for the system.

3.4 PIN Execution Environment

We implemented the de-randomizing execution environment using Intel's dynamic binary instrumentation tool PIN. PIN [26] 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 runtime framework. First, we install a callback that intercepts the loading of all file images. This provides us with the names of all the shared libraries being used, as well as the memory ranges where they have been loaded in the address space. We use the path and name of the library to lookup its key in the database and load it. We save the library's key and memory address range in a hash table-like data structure that allows us to quickly lookup a key using a memory address. The existence of a key in the database also indicates that the library is encoded, so no special handling is required to load system libraries (*i.e.,* not encoded libraries).

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 lookup the key to use for decoding. If the instruction fetched is within the memory range of a shared library we use its key for decoding, or assume no decoding is necessary if no key is present. All instructions not associated with a library are considered to be part of the executable and are decoded using its key. To avoid performing a lookup 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.

3.5 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 scan the address space to identify all memory pages used by PIN. We mark these memory pages by asserting a flag in an array (*page-map*), which 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. At runtime, when additional memory is used by PIN, we update the flags for the newly used pages in the page-map. 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.

Introducing 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 which 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.

3.6 ISR Exceptions

While all instructions in the application are encoded, there are cases where certain external and unencoded instructions need to be executed in the context of the process. For instance, some systems inject code within the stack of a process when a signal is delivered. These *signal trampolines* are used to set up and clean up the context of a signal handler. The instructions are a type of legitimate code-injection performed by the system, and need special handling or their execution will lead to a crash. Fortunately, signal trampolines are very small (approximately 5-7 instructions long), and the instructions used are fixed on every system (*i.e.,* the same instructions are used for all signals in the system). 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.

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`). This object needs to be treated in the same manner as plain shared libraries, allowing the execution of non-randomized code. Since this is a read-only object, we can safely do so.

3.7 Startup Procedure

When a dynamically linked application is executed, the loader looks for shared libraries in certain predefined locations (*e.g.,* "/usr/lib", "/lib", *etc.*), as well as locations specified in the environment (*i.e.,* the environment variable `LD_LIBRARY_PATH`). To enable the loading of the randomized versions of shared libraries, we need to add the shadow folder in the search path. We cannot do so by adding the folder in the system's library search path, as that would cause these libraries to be used instead of the originals for all running applications. Instead, we use `LD_LIBRARY_PATH`. Unfortunately, as PIN itself is dynamically linked we cannot set the variable directly. We employ a wrapper program that we launch using PIN. The wrapper adds the shadow folder in the library search path, and launches the target application, which then looks for libraries in the shadow folder first.

4. 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 in this section 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), 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 encode the binary and 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 1 Gb/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.

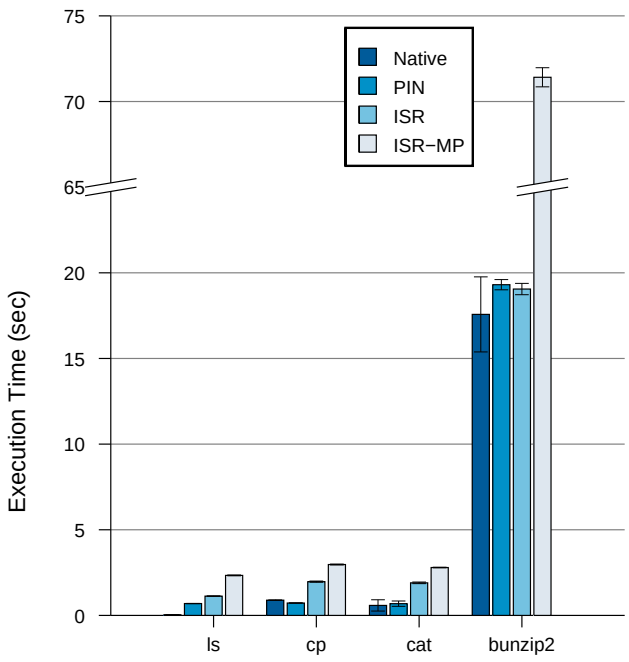

Figure 1: Execution time of basic Linux utilities. The figure draws the mean execution time and standard deviation when running four commonly used Linux utilities.

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.

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 a 1 Gb/s network link to avoid interference with the server.

5. RELATED WORK

Instruction-set randomization was initially proposed as a general approach against code-injection attacks by Gaurav *et al.* [25]. They propose a low-overhead implementation of ISR in hardware, and evaluate it using the Bochs x86 emulator. They also demonstrate the applicability of the approach

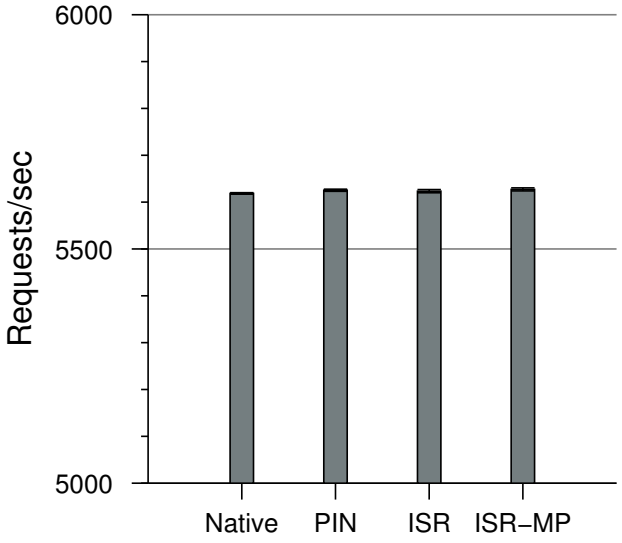

Figure 2: Apache web server throughput. The figure draws the mean reqs/sec and standard deviation as measured by Apache's benchmark utility *ab*.

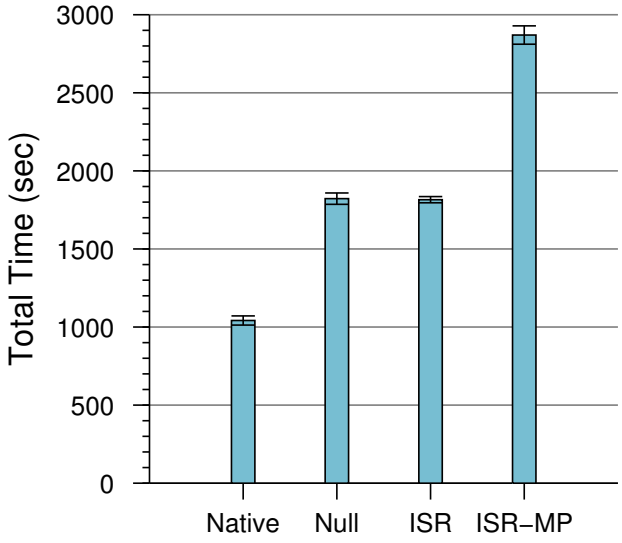

Figure 3: MySQL *test-insert* benchmark. It measures table creation, data insertion, and selection. The figure draws total execution time as reported by the benchmark utility.

on interpreted languages such as Perl, and later SQL [9]. Concurrently, Barrantes *et al.* [4] 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% [18], which makes its use impractical.

The work closest to ours is by Hu *et al.* [24]. They also employ 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, 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 [7, 33]. 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.* [38] 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.* [8] 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 [19], or using techniques like heap spraying [16].

Hardware extensions such as the NoExecute (NX) bit in modern processors [22, 33] 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.

PointGuard [14] 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 include Stackguard [15] and ProPolice [21]. They operate by introducing special secret values in the stack to identify and prevent stack overflow attacks, but can be subverted [10]. Write integrity testing [2] uses static analysis and "guard" values between variables to prevent memory corruption errors, but static analysis alone cannot correctly classify all program writes. CCured [30] 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 datatypes. 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 [31, 17, 13, 23], 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 [6], and in honeypots [37].

6. CONCLUSIONS

We described a fast and practical implementation of ISR based on Intel's dynamic instrumentation tool PIN. Our implementation works on commodity systems, and does not require the recompilation or relinking of target applications. Binaries are randomized at execution time, while shared libraries can be encoded beforehand and shared between the processes executing using ISR. Moreover, we introduce a simple management scheme to keep track of the randomized shared libraries and their associated keys.

Our solution operates with relatively small overhead that makes it an attractive countermeasure to retrofit security sensitive applications with. Applying it on the Apache web server has negligible effect on throughput for static HTML loads, while MySQL performs approximately 75% slower. Furthermore, we show that the overhead is largely attributed to PIN, and can be easily mitigated when applied on long-running I/O driven applications such as network services.

Acknowledgements

This work was supported by the United States Air Force Research Laboratory (AFRL) through Contract FA8650-10-C-7024 and by the National Science Foundation (NSF) through Grant CNS-09-14845. Opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US Government, the Air Force, or the NSF.

7. REFERENCES

[1] Adobe. Security advisory for flash player, adobe reader and acrobat. http://www.adobe.com/support/security/advisories/apsa10-01.html, June 2010.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 263–277, May 2008.

[3] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.

[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. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 281–289, October 2003.

[5] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović. Randomized instruction set emulation. *ACM Trans. Inf. Syst. Secur.*, 8(1):3–40, 2005.

[6] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A tool for analyzing malware. In *Proceedings of the* 15^{th} *European Institute for Computer Antivirus Research (EICAR) Annual Conference*, April 2006.

[7] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceedings of the* 12^{th} *USENIX Security Symposium*, pages 105–120, August 2003.

[8] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the* 14^{th} *USENIX Security Symposium*, pages 255–270, August 2005.

[9] S. W. Boyd, G. S. Kc, M. E. Locasto, A. D. Keromytis, and V. Prevelakis. On the general applicability of instruction-set randomization. *IEEE Transactions on Dependable and Secure Computing*, 99, 2008.

[10] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack*, 5(56), May 2000.

[11] 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, July 2001.

[12] Cert Advisory CA-2003-04: MS-SQL Server Worm. http://www.cert.org/advisories/CA-2003-04.html, January 2003.

[13] M. Costa, J. Crowcroft, M. Castro, and A. Rowstron. Vigilante: End-to-end containment of internet worms. In *Proceedings of the ACM Symposium on Systems and Operating Systems Principles (SOSP)*, October 2005.

[14] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the* 12^{th} *USENIX Security Symposium*, pages 91–104, August 2003.

[15] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the* 7^{th} *USENIX Security Symposium*, January 1998.

[16] DarkReading. Heap spraying: Attackers' latest weapon of choice. http://www.darkreading.com/security/vulnerabilities/showArticle.jhtml?articleID=221901428, November 2009.

[17] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[18] V. Developers. Valgrind user manual – callgrind. http://valgrind.org/docs/manual/cl-manual.html.

[19] T. Durden. Bypassing PaX ASLR protection. *Phrack*, 0x0b(0x3b), July 2002.

[20] C. W. Enumeration. CWE-416: use after free. http://cwe.mitre.org/data/definitions/416.html, April 2010.

[21] J. Etoh. GCC extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/, June 2000.

[22] E. Hardware. CPU-based security: The NX bit. http://hardware.earthweb.com/chips/article.php/3358421, May 2004.

[23] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *Proceedings of the* 1^{st} *ACM EuroSys Conference*, pages 29–41, April 2006.

[24] W. Hu, J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, J. C. Knight, A. Nguyen-Tuong, and J. Rowanhill. 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)*, pages 2–12, June 2006.

[25] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the* 10^{th} *ACM Conference on Computer and Communications Security (CCS)*, October 2003.

[26] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 190–200, June 2005.

[27] M. Conover and w00w00 Security Team. w00w00 on heap overflows. http://www.w00w00.org/files/articles/heaptut.txt.

[28] Microsoft. Microsoft portable executable and common object file format specification. http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx.

[29] D. Moore, C. Shanning, and K. Claffy. Code-Red: a case study on the spread and victims of an Internet worm. In *Proceedings of the* 2^{nd} *Internet Measurement Workshop (IMW)*, pages 273–284, November 2002.

[30] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[31] J. Newsome and D. Song. 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)*, February 2005.

[32] M. Owens. Embedding an SQL database with SQLite. *Linux Journal*, 2003(110):2, June 2003.

[33] PaX Home Page. http://pax.grsecurity.net/.

[34] PCWorld. Dangling pointers could be dangerous. http://www.pcworld.com/article/134982/dangling_pointers_could_be_dangerous.html, July 2007.

[35] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overflows. *IEEE Security & Privacy Magazine*, 2(4):20–27, July/August 2004.

[36] P. Porras, H. Saidi, and V. Yegneswaran. Conficker C analysis. Technical report, SRI International, 2009.

[37] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. In *Proceedings of the* 1^{st} *ACM EuroSys Conference*, April 2006.

[38] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the* 11^{th} *ACM Conference on Computer and Communications Security (CCS)*, pages 298–307, October 2004.

[39] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the* 10^{th} *USENIX Security Symposium*, pages 201–216, August 2001.

[40] A. N. Sovarel, D. Evans, and N. Paul. Where's the FEEB? the effectiveness of instruction set randomization. In *Proceedings of the* 14^{th} *USENIX Security Symposium*, pages 145–160, August 2005.

[41] E. H. Spafford. The Internet worm program: An analysis. Technical Report CSD-TR-823, Purdue University, 1988.

[42] Symantec. Analysis of a zero-day exploit for adobe flash and reader. http://www.symantec.com/connect/blogs/analysis-zero-day-exploit-adobe-flash-and-reader, June 2010.

[43] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, pages 3–17, February 2000.

[44] C. C. Zou, W. Gong, and D. Towsley. Code Red worm propagation modeling and analysis. In *Proceedings of the* 9^{th} *ACM Conference on Computer and Communications Security (CCS)*, pages 138–147, November 2002.