Secure and Practical Defense Against Code-injection Attacks using Software Dynamic Translation

Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W. Davidson, David Evans,
John C. Knight, Anh Nguyen-Tuong, Jonathan Rowanhill
Department of Computer Science
University of Virginia
{wh5a,jdh8d,dww4s,atf3r,jwd,evans,jck,an7s,jch8f}@cs.virginia.edu

Abstract

One of the most common forms of security attacks involves exploiting a vulnerability to inject malicious code into an executing application and then cause the injected code to be executed. A theoretically strong approach to defending against any type of code-injection attack is to create and use a process-specific instruction set that is created by a randomization algorithm. Code injected by an attacker who does not know the randomization key will be invalid for the randomized processor effectively thwarting the attack. This paper describes a secure and efficient implementation of instruction-set randomization (ISR) using software dynamic translation. The paper makes three contributions beyond previous work on ISR. First, we describe an implementation that uses a strong cipher algorithm—the Advanced Encryption Standard (AES), to perform randomization. AES is generally believed to be impervious to known attack methodologies. Second, we demonstrate that ISR using AES can be implemented practically and efficiently (considering both execution time and code size overheads) without requiring special hardware support. The third contribution is that our approach detects malicious code before it is executed. Previous approaches relied on probabilistic arguments that execution of non-randomized foreign code would eventually cause a fault or runtime exception.

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

General Terms: Security

Keywords: Virtual Execution, Software Dynamic Translation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute lists, requires prior specific permission and/or a fee.

VEE '06 June 14–16, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1–59593–332–6/06/0006...$5.00.

1. Introduction

Despite heightened awareness of security concerns, security incidents continue to occur at alarming rates. In 2004, the Department of Homeland Security reported 323 buffer overflow vulnerabilities—an average of 27 new instances per month [13]. The most common attack to exploit a buffer overflow vulnerability is a *code-injection* attack. In a code-injection attack, an attacker exploits a vulnerability, e.g. a buffer overflow, to inject malicious code into a running application and then cause the injected code to be executed. The execution of the malicious code allows the attacker to gain the privileges of the executing program. In the case of programs that communicate over the network, such attacks can be used to break into host systems.

A theoretically strong approach to defending against any type of code-injection attack (irrespective of the vulnerability) is to create and use a process-specific instruction set that is created by a randomization algorithm. Code injected by an attacker who does not know the randomization key will be invalid for the randomized processor thereby thwarting the attack. Such an approach is known as randomized instruction-set emulation (RISE) or instruction-set randomization (ISR) [2, 9]. In this paper, we will use the term ISR exclusively.

The basic operation of an ISR system is as follows. An encryption algorithm (typically XOR'ing the instruction with a key) is applied to an application binary to encrypt the instructions. The encrypted application is executed by an augmented emulator (e.g., Valgrind [17] or Bochs [14]). The emulator is augmented to decrypt the application's instructions before they are executed.

When an attacker exploits a vulnerability to inject code, the injected code is also decrypted before emulation. Unless the attacker knows the encryption key/process, the resulting code will be transformed into, in essence, a random stream of bytes that, when executed, will raise an exception (e.g., invalid opcode, illegal address, etc.).

The security of ISR depends on several factors: the strength of the encryption process, protection of the encryption key, the security of the underlying execution process, and that the decrypted code will, when executed, raise an exception. The practicality of the approach is affected by the overheads in

execution time and space introduced by the encryption and decryption process. This paper describes an implementation of ISR that addresses both the security and practicality issues.

The implementation is secure. It uses the Advanced Encryption Standard (AES) to perform the encryption process. AES has been approved by the United States government to protect classified information at the SECRET level with a 128-bit key and at the TOP SECRET level with either a 192- or 256-bit key [18]. Furthermore, the approach does not require storage of the encryption key on the disk. The key is generated dynamically when the program is loaded. A further benefit is that each execution of an application uses a different key. The underlying execution process is provided by a small, robust virtual execution environment. Finally, the approach does not rely on the generation of an exception or fault by the execution of randomized code. Injected code is detected as it is readied for execution.

To test our implementation, we seeded several published vulnerabilities into several popular server applications and attempted to exploit these vulnerabilities. In all cases, our implementation blocked the vulnerability from being successfully exploited.

The implementation is practical. Rather than use emulation or postulating hardware extensions, we use a robust, efficient software dynamic translation (SDT) system [23]. Performance measurements using a variety of benchmarks including the SPEC CPU2000 suite[1], a domain name server, and a web server, showed the runtime overhead of SDT-based ISR to be modest—17% for SPEC CPU2000, 5–10% for the domain name server, and 2–15% for the web server. Space overhead of SDT-based ISR is also reasonable—the disk image of a protected web server was 77% larger than an unprotected web server. However, the working set size of the ISR implementation was significantly larger than the natively executing application due to the allocation of a 4MB cache for the translated application. More detailed measurements of the overheads of ISR are reported in Section 5.2.

The remainder of this paper is organized as follows. Section 2 describes the class of attacks that ISR handles. Previous work on ISR is described in Section 3. Section 4 describes our SDT-based implementation of ISR. An evaluation of the security and performance of our approach is given in Section 5. Section 6 gives an overview of related work, and Section 7 concludes the paper.

2. Threat Model

The threat model addressed by our infrastructure is application-level binary code injection into an executing program. Attackers exploit some vulnerability in the target program, inject malicious code, and alter program control to execute the malicious code. The model handles all currently identified mechanisms for injecting foreign code into an application (e.g., buffer overflow [20, 13], format string attacks [6], and malloc/free errors [7]). Collectively, these attacks account for over 50% of the CERT advisories issued in the years 1999–2002. Because the approach is independent of the mechanism used to inject code, it can protect against nascent injection mechanisms.

While the threat model covers a wide range of known attacks, those that do not involve code injection are not covered. The model does not cover arc-injection attacks (also known as return-to-libc) [19], or attacks that modify data locations (e.g., a critical data value) [5]. Furthermore, the model assumes that the operating system is secure and that the application image on disk cannot be modified by the attacker.

3. Previous Work

Using randomization to create an instruction set that is unique to the running process so that an attacker cannot create a payload which can be injected into the application and execute properly was first suggested by Thimbleby [27] and later independently developed by groups at the University of New Mexico [3] and Columbia University [9]. Both groups implemented ISR prototypes for the IA-32 instruction set using emulation (Valgrind in the case of New Mexico and Bochs at Columbia).

One of the major differences in the two approaches is how the application code is randomized. Both groups used the XOR operation to produce the randomized binary. The Columbia implementation used a 32-bit key applied to 32-bit blocks containing the instruction or instruction fragment (many x86 instructions are longer than four bytes). The New Mexico implementation used a one-time pad that is the length of the program. The bytes of the one-time pad are XORed with individual bytes of the original application program to create the randomized program. Unfortunately, encryption techniques that use XOR are susceptible to attack. Indeed, it was demonstrated that the New Mexico approach, in some limited situations, can be cracked with modest effort [25].

Because both techniques used emulation, the overhead of decryption and execution was quite high. On CPU-bound benchmarks, the Columbia group reported runtime overhead as high as 25 times native execution speed. On I/O-intensive programs such as *ftp*, the overhead was 1.34x. Based on their results, the Columbia group concluded that ISR would only be feasible with special hardware support.

The New Mexico group carefully benchmarked a single program, *Apache*, and the trend of their results were similar to Columbia's results—I/O-bound programs incur less overhead [2]. When serving many small pages (less than 1KB in size), the runtime overhead was high—2.88x. When serving larger pages (100 KB in size), the runtime overhead was 1.05x. The New Mexico group noted that a software dynamic translator might make ISR practical.

1. We only measured the subset written in C due to the limitation that *Diablo*, the binary rewriter we use, is language dependent. However, the described technique works at the binary level and is independent of any specific language.

Both techniques assumed that the execution of decrypted payloads would eventually cause an exception to occur. Barrantes et al. performed a theoretical analysis of the probability that execution of a sequence of random code will escape (i.e., branch into the application code and not terminate with an error) [2]. The analysis showed that independent of the exploit or process size, there will always be a nonzero probability that the code will escape.

4. Secure and Practical ISR

4.1 Overview

To address the security and performance overheads of the preliminary implementations of ISR, we employ a combination of binary rewriting and software dynamic translation. Binary rewriting is used to prepare the binary for strong encryption and introduce the information necessary to detect foreign code before it is executed. We use an efficient software dynamic translation system to provide the necessary virtual execution environment for safe execution. The SDT system loads and encrypts the application, decrypts the application instructions in preparation for execution, and checks that the decrypted instructions are valid application instructions prior to execution. The current implementation protects IA-32 ELF executables on GNU/Linux. The techniques described should generalize to other systems.

The following subsections describe these two components in more detail. We begin with the virtual execution environment because its operation motivates the necessary transformations performed by the binary rewriter.

4.2 Virtual Execution Environment

We use Strata to provide the virtual execution environment for support of ISR [21, 23]. Strata is a retargetable software dynamic translation infrastructure designed to support experimentation with novel applications of SDT. Strata has been used for a variety of applications including system call monitoring [22], profiling [12], and code compression [24]. The following paragraphs provide a brief introduction Strata's operation.

Strata dynamically loads an application and mediates application execution by examining and translating an application's instructions before they execute on the host CPU (see Figure 1). Strata essentially operates as a co-routine with the application that it is protecting. Translated application instructions are held in a Strata-managed cache called the *fragment cache*. The Strata virtual machine (VM) is first entered by capturing and saving the application context (e.g., program counter (PC), condition codes, registers, etc.). Following context capture, Strata processes the next application instruction. If a translation for this instruction has been cached, Strata transfers control to the cached translated instructions. The logical switching between execution of Strata code and application code is called a *context switch*.

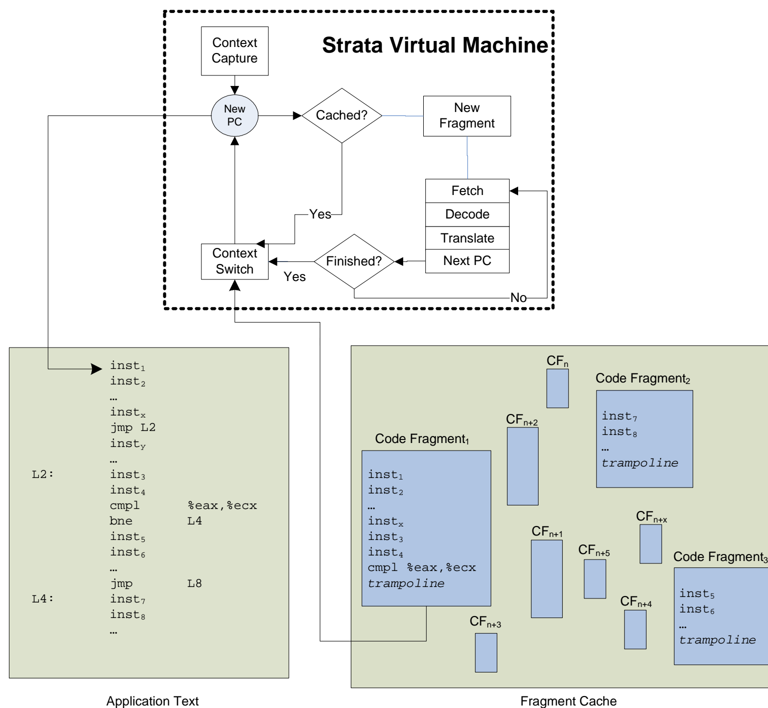


Figure 1: Strata virtual machine virtualizing an application.

If there is no cached translation for the next application instruction, the Strata VM allocates storage in the fragment cache for a new *fragment* of translated instructions. The Strata VM then populates the fragment by fetching, decoding, and translating application instructions one-by-one until an end-of-fragment condition is met. The end-of-fragment condition is dependent on the particular software dynamic translation client being implemented. As the application executes under Strata control, more and more of the application's working set of instructions materialize in the fragment cache.

The implementation of ISR required two simple extensions to Strata. First, we introduced an encryption feature that applies AES to the application text before Strata begins execution of the application. AES is used in electronic codebook (ECB) mode, so each block is encrypted independently. Second, we overrode Strata's default fetch mechanism. The new fetch method decrypts and verifies an instruction before calling the default target-machine fetch method.

Figure 2 gives the basic steps Strata carries out to implement ISR. In Step 1, Strata's security API is enabled to intercept all system calls to `mprotect`. This step prevents an application from inadvertently disabling write protection of the text segment or the fragment cache. In particular, we are concerned with preventing attacks that are intended to corrupt Strata's code since it runs in the same address space as the application.

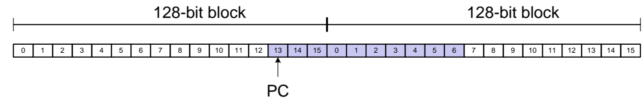
1. Initialize the system call watch table.
2. Encrypt the application.
   a. Obtain a 128-bit encryption key from the pseudo-device `/dev/urandom`.
   b. Use the `mprotect` system call to set write permission for the text segment.
   c. Use the table of address ranges created by the binary rewriter and the key to encrypt the application's text.
   d. Set the text segment permissions to read only.
3. Fetch the next instruction.
   a. Fetch the 128-bit aligned block that contains instruction pointed to by current application PC. Also fetch the next 128-bit aligned block
   b. Decrypt the two 128-bit blocks.
   c. Check that the instruction tag is correct. If the tag is incorrect, report an error and dump the current PC and the plain-text instructions located there.
   d. If the tag is correct, call the default target-machine fetch function to retrieve the next instruction.
   e. The decoding and translation steps proceed as normal.

Figure 2: Runtime decryption and verification.

Step 2 encrypts the binary. To avoid encrypting data or Strata's code, the rewriting process creates and embeds a table in the application text, called the `encrypttable`, that specifies the blocks of the application text that should be encrypted. The binary rewriter also modifies the application text so that the start of each block is aligned on a 128-bit address boundary. Strata uses the `mprotect` system call to enable modification of the text segment. Using the information in the `encrypttable` and a 128-bit key obtained from the pseudo-device `/dev/urandom`, Strata encrypts the application text. The text segment permissions are then set to read only. The impact on demand paging caused by walking the full text segment is amortized over the execution of the application.

Step 3 describes the modification necessary to decrypt and verify application instructions. The new fetch method loads two 128-bit blocks into a decoding buffer. It fetches the block that contains the first byte of the instruction pointed to by the PC and the following 128-bit block. Both blocks are then decrypted. Fetching two consecutive 128-bit blocks guarantees that the complete instruction is fetched and decrypted even if the instruction starts on the last byte of the first 128-bit block (the maximum length of an IA-32 instruction is 15 bytes).

To illustrate the process, suppose the PC points to a ten-byte instruction that begins at memory location `0x1017B3D`. The decryption engine fetches and decrypts the 128-bit blocks at addresses `0x1017B30` and `0x107B40`. The following is a schematic of the decoding buffer after the fetches.



The shaded portion indicates the bytes of the buffer that contain the ten-byte instruction.

As part of the binary rewriting process (see Section 4.3), our ISR implementation tags each instruction with a simple eight-bit code (called an instruction tag). These tags permit Strata to identify injected code during its translation step. After decrypting the two blocks, Strata checks the instruction tag to ensure that the fetched bytes represent a valid application instruction. If the instruction tag is valid, Strata simply invokes the default fetch method with the PC pointing at the first byte of the instruction.

While an attacker could guess the instruction tag code, they cannot know the encryption key and therefore they cannot construct a payload that will be correctly tagged after decryption. Thus, if an instruction tag is invalid, the first stage of a code injection attack is underway—a vulnerability has been exploited to inject code and control flow has been diverted in an attempt to execute the malicious code. When Strata detects an invalid instruction tag, it reports the violation, and dumps the current program counter and the undecrypted code pointed to by the program counter (i.e., the malicious payload). This information can be used for offline forensic analysis.

The use of an eight-bit tag means that there is a 1 in 256 chance that a tag is coincidentally correct. However, for

Strata to execute the fragment containing the injected code, the tag for each instruction in the fragment must be correct. Thus for a fragment containing four instructions, the probability that four tags would be coincidently correct is 2^{-32}.

It is important to note that the process of decrypting the application text, checking the instruction tag, and building a specific fragment generally only occurs once. Thus, the performance overhead of SDT-based implementation of ISR is closely related to the basic overhead of software dynamic translation. Section 5.2 provides detailed measurements of the overheads of our SDT-based implementation of ISR.

There are a few other details that deserve discussion. Strata controls access to the fragment cache using the `mprotect` system call. During application execution, the fragment cache is write protected. The decryption key is also maintained in a memory region not accessible to the application. On a context switch from the application to Strata, Strata makes the fragment cache writeable so that it may create new fragments or perform updates of existing fragments. It also makes the location containing the decryption key readable. Before the context switch back to the application occurs, Strata reprotects the fragment cache and the encryption key.

A problem worth mentioning as we implemented ISR is the shared library problem. Strata, which is written in C and some assembly language, uses a few `libc` routines which may also be used by an application program. This sharing of code creates a problem. When Strata translates these routines as part of the virtualization process of the application, it expects these routines to be encrypted. However, when Strata uses these routines, they should be plain text.

To address this problem, we used `objcopy`, an ELF manipulation tool, to produce a separate copy of all the `libc` functions used by Strata. For example, a `printf` call in Strata will be linked to `strata_libc_printf`. In contrast, a `printf` call in the application's code is linked to the normal `libc`, which is encrypted at run time. This space overhead from this duplication is minimal because Strata uses very few `libc` routines. In the future, we plan to eliminate code sharing by making Strata independent of any standard library.

Our current implementation of Strata does not support applications that employ legitimate uses of self-modifying code. We do not view this as a serious limitation. None of the critical applications that we have examined employ self-modifying code. Nonetheless, we plan to investigate techniques for safely handling legitimate uses of self-modifying code in the future.

4.3 Binary Preprocessing

To prepare the binary for encryption using AES and to introduce the necessary instruction tags, we modified *Diablo* 0.3, an existing binary rewriting tool [4].

Figure 3 illustrates the basic workflow of *Diablo*. *Diablo* reads all object files and libraries constituting an application, the linked application, and the map file generated by the normal linker. In phase 1, *Diablo* uses this information to replay the linking process of the normal linker. and translate the object files and libraries into an internal representation. The linker map file provides the information *Diablo* needs to generate the same code and data layout as the original binary.

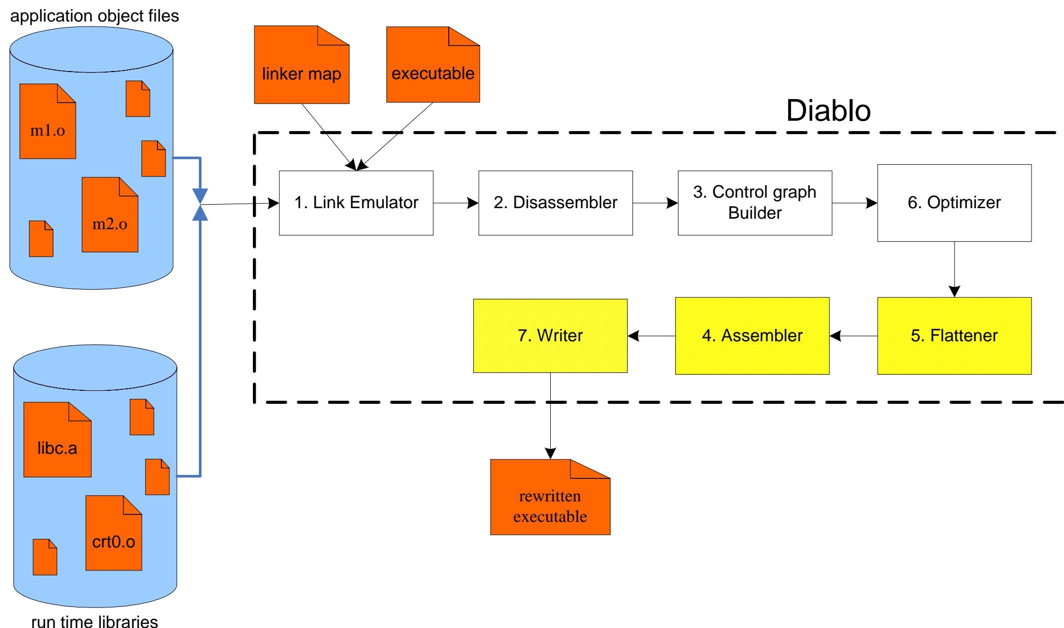


Figure 3: Work flow of the binary rewriter *Diablo* (Version 0.3).

Phase 2 disassembles the instructions and phase 3 builds a control flow graph (CFG). Phase 4 then applies various analysis and optimization techniques to the CFG (e.g., useless code elimination, architecture-dependent peephole optimizations, etc.). After completion of phase 4, phase 5 flattens the CFG into a linear representation and phase 6 produces target-machine instructions. In the final step, the binary writer emits the modified executable.

The shaded blocks in Figure 3 are the *Diablo* modules that required extension to produce a binary with the transformations and informations needed to support ISR. The extensions to each phase are outlined in Figure 4.

```
1.  Flattener
    For each basic block do:
    a.  Determine the source of the basic block. If the
        basic block is application code, mark the basic
        block for encryption. Otherwise do not mark
        the block for encryption.
    b.  If the block is marked for encryption and
        instruction tagging is enabled, reserve one byte
        before each instruction for the instruction tag.
    c.  Recalculate the offsets among basic blocks and
        update all instructions affected.
    d.  Maintain a record of each block that should be
        encrypted.
2.  Assembler
    For each instruction do:
    a.  Determine the source of the instruction.
    b.  If the instruction is application code and
        instruction tagging is enabled, insert the tag
        before it.
3.  Writer
    a.  Create a new section, encrypttable, to contain
        the information about the text blocks to
        encrypt at load time.
    b.  Set up the ELF executable and output the
        binary (the text section, the data section, the
        encrypttable section, and any other sections).
```

Figure 4: *Diablo* extensions to support ISR.

Phase 5, the Flattener, assigns a linear order to the CFG and updates the offsets in control transfer instructions according to that order. We added a function `AlignBlock` that is invoked after the linear order is assigned, but before offsets are updated. This function processes each basic block. If the block is application code, `AlignBlock` reserves space for a instruction tag before each instruction. It then aligns the block appropriately by padding the beginning of the block with NOPs.

Not every basic block needs alignment. If the previous block was aligned and the following basic block is part of the application text, it can be grouped with the previous basic block. Padding only happens on the boundary between Strata and application code. After all blocks are processed, the Flattener recalculates branch offsets and updates all instructions affected. The starting and ending address of each block that should be encrypted is collected so this information can included in the modified binary emitted by the Writer.

Phase 6, the Assembler, is modified to fill the placeholder preceding each instruction with a tag if instruction tagging is enabled.

Diablo's final phase emits the modified binary to disk. This phase was extended to create a new section, `.encrypttable`, that contains the starting and ending address of each block that should be encrypted.

5. Evaluation

With any system designed to protect software against malicious exploitation of vulnerabilities, there are tradeoffs in terms of performance and the level of security provided. In this section, we evaluate the security and performance of SDT-based ISR.

5.1 Security Evaluation

As previously described, our implementation uses AES to encrypt the application text using a key that is generated at runtime. It is generally believed that AES is secure [18].

When encryption is used to protect a system, an important issue is management of the encryption key. Where is the key stored? How is the key protected? How long is the key valid? Our approach addresses these issues. The encryption key is never stored on disk, the key is maintained in a protected region of memory only accessible by Strata, and a new key is generated for each execution of the application.

To evaluate the security of SDT-based ISR, we seeded published vulnerabilities and synthetic vulnerabilities into several real applications and then exploited the vulnerability to effect a code-injection attack. Table 1 lists the applications, the type of vulnerability, and the target memory region of the injected code. The column labeled BID gives the Bugtraq ID for the vulnerability. An N/A in this column indicates a synthetic vulnerability. For each vulnerability, we demonstrated that exploitation of the vulnerability could be used to compromise an unprotected system. In all cases, ISR detected the attempt to execute injected code and prevented the attack from proceeding.

Table 1: Tested applications.

Application	Vulnerability	Location	BID
Apache	Buffer overflow	Heap	5363
Apache	Format string	Heap/Stack	N/A
Samba	Buffer overflow	Stack	7106
BIND	Format string	Heap/Stack	788
rpc.statd	Format string	Global offset table	1480
cvs server	Double free	Stack	6650

5.2 Performance Evaluation

A major concern raised by initial implementations of ISR was the high runtime overheads incurred. To evaluate the runtime overhead of SDT-based ISR, we measured the performance of a variety of benchmarks. In all measurements, the performance measures are normalized to native execution—the application running directly on the hardware.

All measurements were performed on a dual processor 1GHz P3 system with 1GB of RAM. When an application required both a client and a server, the server executed on the P3, while all clients executed on 2.8GHz P4 systems with 512MB of RAM. The server used two 100 Base-T Ethernet ports and each client used a single 100 Base-T NIC. All systems ran Red Hat 7.3 with the P3 and the P4 systems running 2.4.13 and 2.6.11 Linux kernels, respectively. Hyperthreading was enabled on the P4's, whereas the P3 system was booted with a uniprocessor kernel to prevent the second CPU from being used during the measurements. For all performance experiments, the size of Strata's fragment cache was fixed at 4MB.

The asymmetry between the server and client configurations was motivated by our desire to measure the processor overhead imposed by the Strata VM. Providing the server twice as much network and about a sixth as much processing power relative to the clients allowed us to drive the server to near full CPU utilization under most testing conditions. During these tests the network was not saturated from the server's perspective, and therefore did not obscure the CPU costs behind the network.

Figure 5 shows the performance results for SPEC CPU2000. We measured the overheads of the baseline SDT system (no ISR) and the SDT-based ISR system. The performance metric used to compute the overhead was the reportable SPEC ratio produced by the SPEC measurement infrastructure.

The average overhead for SDT-based ISR is 1.17 while the overhead of the baseline SDT system is 1.16. This basic trend is seen for all the benchmarks—ISR incurs little or no additional overhead over a baseline SDT system.

It is interesting to note that the high average is due to a few outliers—*perlbmk*, *gap*, and *gcc*. These benchmarks execute a high percentage of indirect control transfers which are problematic for SDT systems [15, 23].

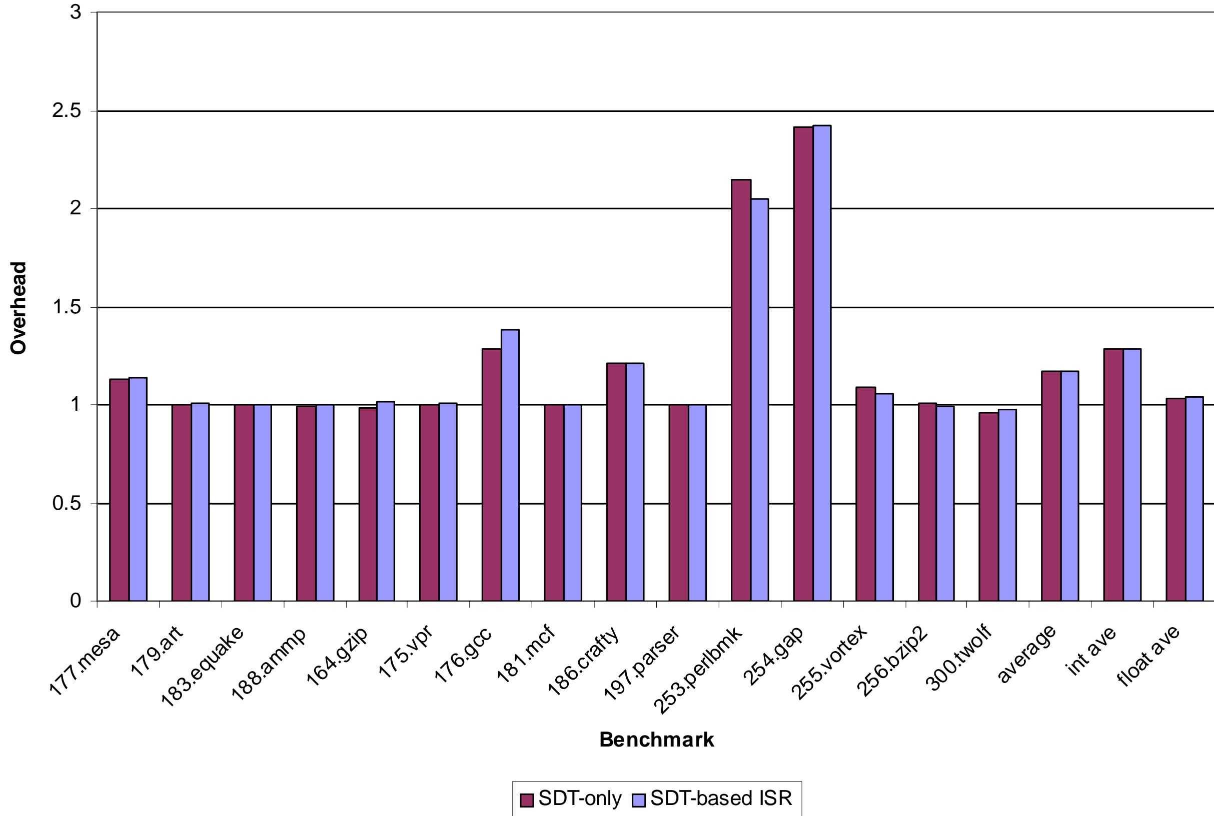


Figure 5: SDT overhead and SDT-ISR overhead normalized to native execution. Metric: SPEC ratio.

We also measured the overhead of two applications that are representative of the types of programs that might be desirable to protect with ISR. One is *Apache*, the widely-used Web server. The other is *BIND*, a widely used domain name server.

To measure *Apache* performance, we used *httperf*, a web server performance measurement tool. *Httperf* collects and reports a large number of performance related metrics. In our case we chose to use the elapsed time for completing a fixed amount of work.

The native version of *Apache* was processed by *Diablo* so that comparisons across all binaries were consistent in this respect. Technically, only the ISR enabled versions of Strata require this step.

Approximately 400MB of traffic was generated by each client during each test. Each test was run three times. To determine if performance was sensitive to the size of the page served, we measured performance using a variety of page sizes. Four clients were run with two accessing the server on each of its network interfaces to balance the network load. *Apache* was configured to pre-fork one server for each test client.

Since we wanted to minimize the interference of the network overhead, each client used a single connection per run and pipelined all requests using the keep-alive request option. This eliminated the latency and network traffic from establishing and tearing down many TCP connections. This network overhead would cause the server to become idle while saturating the network and thereby hide nearly all of the Strata processor overhead. Figure 6 contains the results.

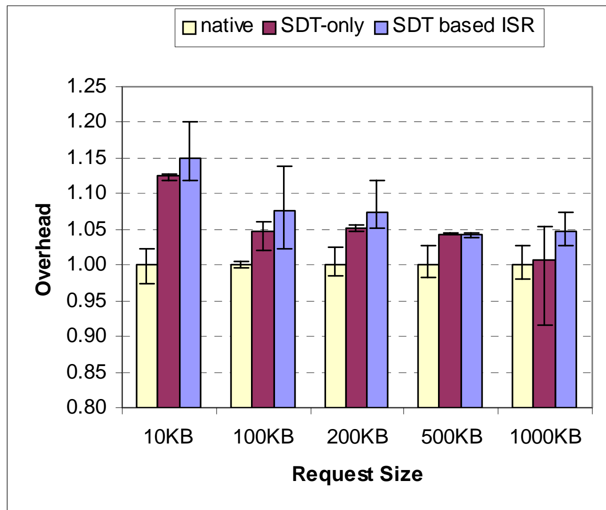


Figure 6: *Apache* overhead normalized to native execution. Metric: client requests served per second.

As the chart shows, the overhead of running either the baseline SDT system or the SDT-based ISR system varies from a few percent to 15 percent. The error bars show that there was considerable variability in performance across the three runs even for the natively executing server. We believe the variability is due to the inherently inevitable fluctuations when measuring client and server processes communicating over a network.

To measure the performance of the Berkeley Internet Name Domain server, *BIND*, we created three representative zone files. Briefly, a zone file contains resource records for mapping names such as `www.apache.org` to an IP address, and for mapping an IP address to a name (a reverse lookup). We created zone files containing 1000 records, 10,000 records, and 100,000 records to represent a small organization, a mid-size organization, and a large organization, respectively. Using *queryperf*, a DNS server performance testing tool, we measured the number of queries processed per second of *BIND* running under our SDT system with and without ISR enabled.

The performance measurement results are presented in Figure 7. The overhead of querying the small and mid-size databases is about 10%, while the overhead for the larger database was 5%. Again, there was no statistically significant difference between the SDT-only system and the SDT-based ISR system.

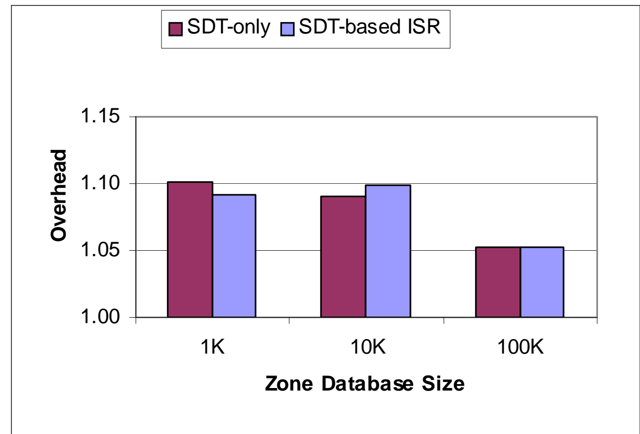


Figure 7: *BIND* overhead normalized to native execution. Metric: queries per second.

We also measured the size overhead of *BIND* and *Apache*. First, we measured the size of the SDT-only and SDT-based ISR disk images. The results for the disk image overhead is presented in Table 2. For *BIND* and *Apache*, the disk image size of the SDT-based ISR binary was 51 and 77 percent larger, respectively, than the text size of the corresponding native binaries. As the table shows, most of the size overhead is due to the growth of the text segments due to instruction tagging. We also examined the size of `encrypttable` and found it to be negligible.

We also observed the resident set size of the running applications. These measurements are presented in Table 3. As can be seen by comparing the resident set size of the native exe-

Table 2: Disk image overhead (Kilobytes).

	BIND				Apache			
	Native	**SDT-only**	**SDT-based ISR**	**SDT-based ISR Expansion**	**Native**	**SDT-only**	**SDT-based ISR**	**SDT-based ISR Expansion**
Disk image	1811	1872	2731	1.51x	916	987	1617	1.77x
text	1786	1838	2690	1.51x	875	939	1566	1.79x
data	23	28	32	1.39x	34	39	43	1.26x
bss	13	32	41	3.15x	166	186	194	1.17x

Table 3: Resident set size overhead (Megabytes).

Zone File Size (BIND) or Web Page Size (Apache)	BIND				Apache			
	Native	**SDT-only**	**SDT-based ISR**	**Expansion**	**Native**	**SDT-only**	**SDT-based ISR**	**Expansion**
1K	1.6	5.7	7.0	4.38x	N/A	N/A	N/A	N/A
10K	3.5	7.5	8.9	2.54x	0.7	3.0	3.6	5.14x
100K	22.0	26.0	27.4	1.24x	0.7	3.0	3.6	5.14x
1000K	N/A	N/A	N/A	N/A	0.7	3.0	3.6	5.14x

cution to SDT-only execution for both *BIND* and *Apache*, the growth is consistent with the addition of a 4 MB fragment cache. The difference between the resident set size of the SDT-only execution and SDT-based ISR execution is due to the instruction tags and duplication of library functions (see Section 4.2).

For BIND, the larger zone files greatly affected the resident set size. This is not surprising since BIND builds a red/black tree of the zone information in memory for quick lookups.

While we believe that the size overheads are reasonable for critical server applications, for some environments reducing space overhead may be desirable or necessary. The size of the text segment could be substantially reduced by computing a secure hash function for a block of instructions rather than a one-byte instruction tag for every instruction. Furthermore, rather than fixing the size of the fragment cache, its size could be adjusted for the particular application.

6. Related Work

Code injection attacks represent a major threat to computer security, and as a result there is a large body of work describing various techniques for stopping attackers from running injected code. Many of these techniques focus on particular areas of memory that are often attacked, most often the stack. StackGuard [8] and PaX [26] are two popular example of such methods.

Previous work involving software-base implementations of ISR is described in Section 3. Milenkovic et al. propose a method of basic block signing, similar to ISR, but partially implemented in hardware [16]. This system uses AES, which a hardware key to create a signature of each basic block to ensure that it has not been modified. Similarly Kirovski et al. created the "Secure Program Execution Framework" for the ARM instruction-set architecture [11]. This framework also creates hashes of groups of instructions, which are checked in hardware before the instructions are allowed to execute. However, the system constructs the hashes in such a way that instruction rescheduling and basic block reordering, and register permutations could still be performed.

Software dynamic translators have also been used for other security systems, mainly in policy enforcement. Strata has been used to enforce security policies [22]. Here Strata provides an API to watch sensitive system calls and function calls, and alter them or prevent them if they behave outside the implemented policy.

Because of the large number of code injection attacks, there has been much research focused on defending against such attacks. Some early work focused solely on preventing stack-smashing attacks [8, 13, 22]. As other types of attacks were developed, other approaches for preventing exploitation of code vulnerabilities were proposed.

Abadi et al. propose restrictions on control flow using static binary rewriting [1]. This system uses labels to ensure that return instructions match valid return sites. DynamoRIO is used as the base for program shepherding [10]. Program shepherding restricts program execution based on a number of policies like disallowing modified code and restricting targets of branch instructions.

Both program shepherding and ISR prevent code injection attacks by enforcing policies about what code can be exe-

cuted. The mechanism used in ISR is simple and comprehensive—only code from the original application can be executed.

Recently, hardware vendors and operating system designers have begun to provide memory protection primitives that disallow execution from writeable memory. This mechanism provides similar protection to ISR via hardware and the operating system. Clearly, if such mechanisms exist, they should be used.

However, there is an important class of processors that do not have memory management hardware or protection mechanisms—processors for embedded systems. Because embedded processors are often used for critical services and they are increasingly on the network (e.g., SCADA systems, some medical devices, transportation control components, some military systems etc.), protecting them from attack is becoming increasingly important. For such systems, an efficient software-based ISR system could be useful. However, it is worth noting that many embedded systems have size constraints and an important research issue for using ISR in embedded systems is reducing the size overhead of ISR.

7. Summary

This paper has described a software dynamic translation-based implementation of instruction-set randomization. Instruction-set randomization is a powerful technique that defends against all application-level binary code injection attacks independent of the vulnerability used to inject the code. The implementation uses a strong encryption algorithm, the Advanced Encryption Standard, to produce a random instruction set each time the protected application is loaded and executed. Without access to the encryption key, an adversary cannot produce a payload that will successfully execute on a protected system. We tested the security of our system by seeding different types of vulnerabilities into applications and then exploiting the vulnerabilities to inject code. In every case, our ISR-protected implementations detected and prevented execution of the foreign code.

The SDT-based implementation of ISR is sufficiently efficient to be used to protect critical server applications that are often the target of attack. Measurements of an ISR-protected *Apache* web server showed performance loss of only 5 to 15 percent over a natively executing *Apache* web server. Similarly, the performance of an ISR-protected domain name server was evaluated. The performance loss over a natively executing version was observed to be between 5 and 10 percent.

These performance results along with the security of the approach make SDT-based ISR a viable protection mechanism for critical server applications. While the approach only protects against code-injection attacks, these represent a large class of attacks. Encouraged by the performance results, we are investigating the use of SDT to protect against other types of attack including arc injection and data corruption attacks.

8. Acknowledgements

This research was supported by DARPA under agreement number FA8750-04-2-0246 and the National Science Foundation under grants CNS–0305144 and CNS–0524432. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

9. References

[1] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow integrity. In *Microsoft Research Technical Report MSF-TR-05-18* (2005).

[2] Barrantes, E. G., Ackley, D. H., Forrest, S., and Stefanovic, D. Randomized instruction set emulation. *ACM Transactions on Information System Security. 8*, 1 (2005), 3–40.

[3] Barrantes, E. G., Ackley, D. H., Palmer, T. S., Stefanovic, D., and Zovi, D. D. Randomized instruction set emulation to disrupt binary code injection attacks. In *CCS '03: Proceedings of the 10th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2003), ACM Press, pp. 281–289.

[4] Bus, B. D., Sutter, B. D., Put, L. V., Chanet, D., and Bosschere, K. D. Link-time optimization of ARM binaries. *ACM SIGPLAN Notices 39*, 7 (July 2004), 211–220.

[5] Chen, S., Xu, J., Sezer, E., Gauriar, P., and Iyer, R. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium* (Berkeley, CA, USA, 2005), USENIX Association, pp. 177–192.

[6] Cowan, C., Barringer, M., Beattie, S., and Kroah-Hartman, G. Formatguard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium* (August 2001).

[7] Cowan, C., Beattie, S., Johansen, J., and Wagle, P. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium* (Aug. 2003), USENIX, pp. 91–104.

[8] Cowan, C., Pu, C., Maier, D., Hinton, H., Bakke, P., Beattie, S., Grier, A., Wagle, P., and Zhang, Q. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 1998 USENIX Security Symposium* (January 1998).

[9] Kc, G. S., Keromytis, A. D., and Prevelakis, V. Countering code-injection attacks with instruction-set randomization. In *CCS '03: Proceedings of the 10th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2003), ACM Press, pp. 272–280.

[10] Kiriansky, V., Bruening, D., and Amarasinghe, S. P. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium* (Berkeley, CA, USA, 2002), USENIX Association, pp. 191–206.

[11] Kirovski, D., Drinic, M., and Potkonjak, M. Enabling trusted software integrity. In *ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2002), ACM Press, pp. 108–120.

[12] Kumar, N., and Childers, B. Flexible instrumentation for software dynamic translation. In *Workshop on Exploring the Trace Space, International Conference on Supercomputing* (2003).

[13] Kuperman, B. A., Brodley, C. E., Ozdoganoglu, H., Vijaykumar, T. N., and Jalote, A. Detection and prevention of stack buffer overflow attacks. *Communications of the ACM 48*, 11 (2005), 50–56.

[14] Lawton, K. P. Bochs: A portable pc emulator for Unix/X. *Linux J. 1996*, 29es (1996), 7.

[15] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2005), ACM Press, pp. 190–200.

[16] Milenkovic, M., Milenkovic, A., and Jovanov, E. Using instruction block signatures to counter code injection attacks. *SIGARCH Computer Architecture News 33*, 1 (2005), 108–117.

[17] Nethercote, N. Dynamic binary analysis and instrumentation. Tech. Rep. UCAM-CL-TR-606, University of Cambridge, Computer Laboratory, Nov. 2004.

[18] The Committee on National Security Systems, T. C. National policy on the use of the advanced encryption standard (AES) to protect national security systems and national security information. Tech. rep., National Security Agency, 2003.

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

[20] Prasad, M., and Chiueh, T. A binary rewriting defense against stack-based buffer overflow attacks. In *Proceedings of the 2003 USENIX Annual Technical Conference* (June 2003), pp. 211–224.

[21] Scott, K., and Davidson, J. Strata: A software dynamic translation infrastructure. In *IEEE Workshop on Binary Translation* (September 2001).

[22] Scott, K., and Davidson, J. W. Safe virtual execution using software dynamic translation. In *Proceedings of the 18th Annual Computer Security Applications Conference* (Las Vegas, NV, December 2002), pp. 209–218.

[23] Scott, K., Kumar, N., Velusamy, S., Childers, B. R., Davidson, J. W., and Soffa, M. L. Retargetable and reconfigurable software dynamic translation. In *International Symposium on Code Generation and Optimization* (San Francisco, CA, Mar. 2003), IEEE Computer Society, pp. 36–47.

[24] Shogan, S., and Childers, B. Compact binaries with code compression in a software dynamic translator. In *Design Automation and Test in Europe* (2004).

[25] Sovarel, N., Evans, D., and Paul, N. Where's the feeb? the effectiveness of instruction set randomization. In *Proceedings of the 14th USENIX Security Conference* (2005).

[26] The PAX Team. http://pax.grsecurity.net.

[27] Thimbleby, H. Can viruses ever be useful? *Computers and Security 10*, 2 (1991), 111–114.