AFRL-IF-RS-TR-2007-9
Final Technical Report
January 2007

GENESIS: A FRAMEWORK FOR ACHIEVING SOFTWARE COMPONENT DIVERSITY

University of Virginia

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. S472

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

AFRL-IF-RS-TR-2007-9 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/	/s/
MICHAEL J. HENSON, Capt, USAF Work Unit Manager	WARREN H. DEBANY, Jr. Technical Advisor, Information Grid Division Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JAN 2007	Final	Jun 04 – Aug 06

4. TITLE AND SUBTITLE

GENESIS: A FRAMEWORK FOR ACHIEVING SOFTWARE COMPONENT DIVERSITY

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA8750-04-2-0246

5c. PROGRAM ELEMENT NUMBER
62301E

6. AUTHOR(S)

J.C. Knight, J.W. Davidson, D. Evans, A. Nguyen-Tuong and C. Wang

5d. PROJECT NUMBER
S472

5e. TASK NUMBER
SR

5f. WORK UNIT NUMBER
SP

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Virginia
151 Engineers Way
Charlottesville VA 22904-4740

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

AFRL/IFGB
525 Brooks Rd
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2007-9

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 07- **017**

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The Genesis project sought to provide security through the diversification of software. A major weakness with current information systems is that they use software applications that are clones of each other; a major exploitable flaw in one implies a flaw in all other similarly configured software packages. Breaking this software monoculture was the goal of the bio-inspired diversity area of DARPA's self-regenerative systems program. The Genesis project exceeded the program's goal of producing 100 functionally-equivalent versions of software such that no more than 33 exhibited the same deficiency. This report presents an overview of the Genesis project, the current status of the Genesis Diversity Toolkit, and future opportunities for technical transfer and research.

15. SUBJECT TERMS
Cyber Operations, Information Warfare, Information Assurance, Software Diversity, Monoculture

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Capt Michael Henson
U	U	U	UL	119	19b. TELEPHONE NUMBER (*Include area code*)

Table of Contents

List of Figures

1 Introduction

The overall goal of phase I of the Self-Regenerative System (SRS) program (DARPA BAA 03-44) was to develop technology for building military computing systems that could provide critical functionality at all times, in spite of damage caused by unintentional errors or attacks.

A major problem today is that of our software monoculture. Critical infrastructure software applications such as web servers, database servers, routers, and name resolution servers to name only a few, are all shipped identically. An exploitable vulnerability present in one deployed software application strongly implies an exploitable flaw in all copies of that application. This situation provides adversaries with an overwhelming advantage and is very serious because it multiplies the impact of any vulnerability by the number of machines running the software that contains the vulnerability. Once a vulnerability is exposed, adversaries seek out machines that are using the software with which the vulnerability is associated and proceed to exploit the vulnerability. Thus, the software monoculture enables the spread of both worms, i.e., self-replicating malicious code, and attacks that target specific servers.

Drawing inspiration from biological systems in which genetic diversity provides immunity against a broad range of disease, the Genesis project sought to reproduce the genetic diversity found in nature by deliberately and systematically introducing diversity into software components. The basic idea was that while the phenotype (functional behavior) of software components would be similar, the resulting genotypes would contain enough variations to protect software applications against a broad class of attacks, including both self-replicating and directed attacks.

In the past, the application of diversity for critical systems has been severely limited by the fact that creating diverse versions has been attempted, for the most part, by producing the versions using traditional, resource intensive methods. Creating two diverse web servers, for example, involved actually writing both implementations. Clearly, this approach would not yield a large number of diverse versions unless unrealistic amounts of resources were available. The Genesis project sought machine transformation techniques to automate the task of creating large number of program variants.

The success metric as specified in the SRS program was that of automatically producing 100 diverse but functionally equivalent versions of a software component such that no more than thirty-three versions of a component shared the same deficiency. We exceeded this goal through the use of novel program transformation techniques coupled with advances in virtual machine technology, with demonstrated good performance on a range of real-world and critical applications.

2 Genesis Overview

In the Genesis approach, we took a biologically inspired approach to diversity in which we investigated the two fundamental aspects of computation, state and state change, and we introduce diversity systematically and comprehensively to both. In practice by "state" we mean the data upon which a computation operates and by "state change" we mean the changes effected by some interpreter (a hardware entity or a software interpreter) in response to a set of instructions. We took a very general view of these two notions so that some entities were viewed as part of a state at one point and as being involved in state change at a different point. For example, machine instructions were part of the operating state of a compiler, i.e., data, but they controlled an interpreter during program execution, i.e., instructions. Furthermore, we took a multi-hierarchical and composable view of diversity in which we combined transformations from different phases of a program's lifecycle, from compile-time all the way to execution-time.

The Genesis project was implemented as the *Genesis Diversity Toolkit* henceforth called the *GDT*. The GDT was a collection of compile-time, link-time, run-time, and post-processing tools that allowed diversification of C and C++ software. The Genesis toolkit included the following components:

- Zephyr, a compiler infrastructure developed at the University of Virginia.
- Diablo, an open source static binary rewriter developed at Ghent University in Belgium.
- Strata, an application-level virtual machine developed at the University of Virginia, along with several modules to effect dynamic diversity techniques.

2.1 Genesis Diversity Techniques

The GDT supported the following diversity techniques:

- *Address Space Randomization (ASR)*. ASR was a link-time option, whereby the static (uninitialized and initialized) data segments were offset by a random amount. This coarse-grained technique obfuscated the location of critical variables.
- *Stack Space Randomization (SSR)*. This technique randomized the padding between stack frames.
- *Simple Execution Randomization (SER)*. This technique used a simple XOR encoding of a binary executable. This was mainly a proof-of-concept implementation that has been deprecated by the development of Strong Instruction Set Randomization.
- *Strong Instruction Set Randomization (SISR)*. This technique protected applications against both known and unknown code-injection attacks.
- Calling Sequence Diversity (CSD). This technique modified the calling convention of functions to incorporate a hidden extra argument whose value is both generated at run-time and dependent on the history of the calling context. This technique defended against return-to-libc attacks [Nergal01].

The GDT provided defense-in-depth by allowing application developers to select and compose among various techniques. Note that the first three techniques, ASR, SSR, and SER provided only a limited amount of entropy relative to SISR and CSD. However, attack code tends to be fragile and even small perturbations in the execution environment will thwart attacks.

Figure 1 illustrates the various configuration options for the Genesis toolkit. Developers could compose various techniques, specify various configuration parameters, and generate an arbitrary number of software variants. In practice, these various options were set via standard build scripts, e.g., makefiles.

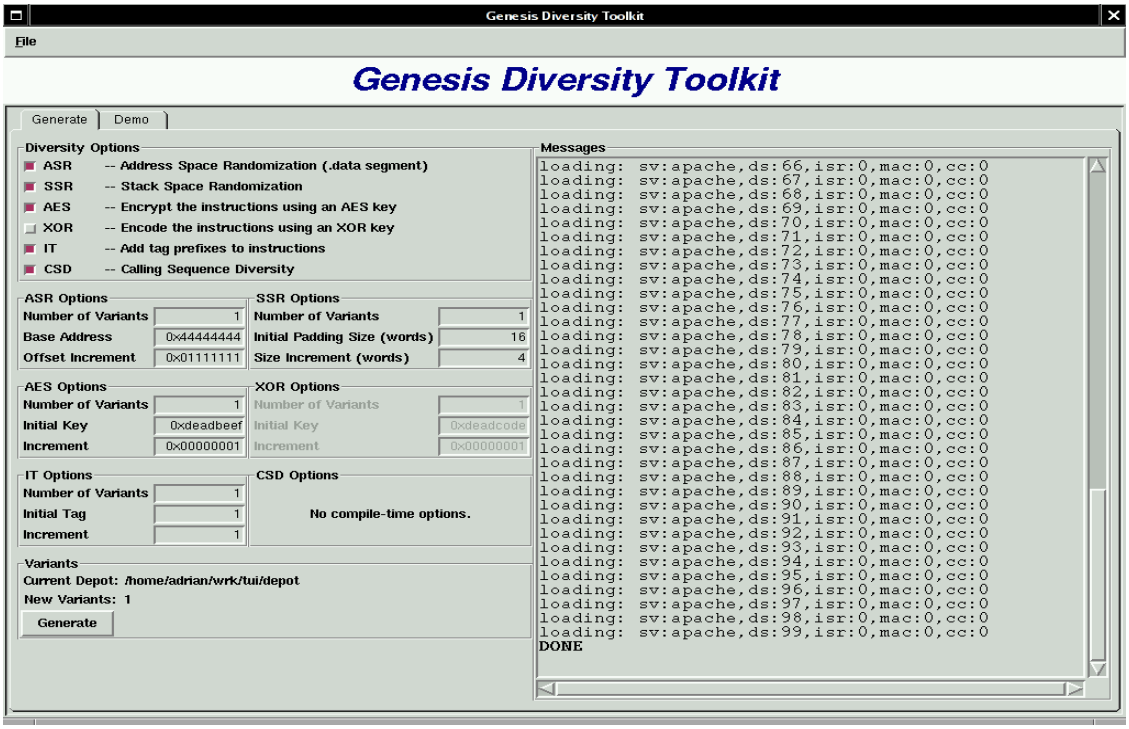


Figure 1. Genesis Diversity Toolkit Configuration Panel

Next we provide an overview of the Strata Virtual Machine and its role in the implementation of *Strong Instruction Set Randomization* and *Calling Sequence Diversity*.

2.2 Genesis: Strata Virtual Machine

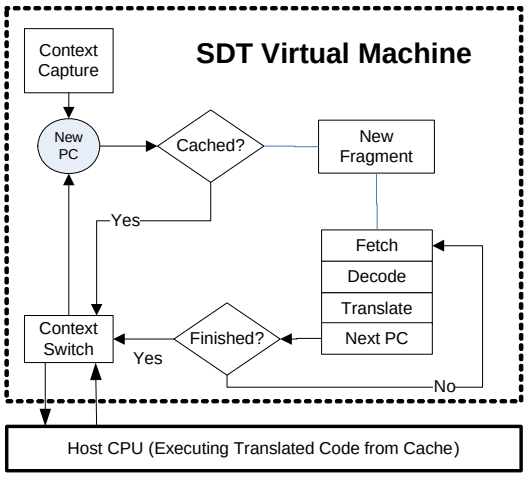


Figure 2. Strata Virtual Machine Architecture

At the core of our approach was *Strata*, a software dynamic translator (SDT) that implemented an application-level virtual machine. Strata was a small, efficient run-time execution environment that hosted, monitored and ran applications. Strata could affect an executing program by injecting new code, modifying some existing code, or controlling the execution of the program in some way.

Strata dynamically loads an application and mediates application execution by examining and translating an application's instructions before they execute on the host CPU (Figure 2). 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, a context switch restores the application context and begins executing cached translated instructions on the host CPU.

In the case of the GDT, Strata was used to support important run-time features of software diversity, including dynamic code encryption/decryption and calling sequence diversity.

2.3 Strong Instruction Set Randomization

We provide a general overview of Instruction Set Randomization (ISR). A detailed description is provided in Appendix A.

The main idea behind ISR for 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 thereby thwarting the attack. Such an approach is known as randomized instruction-set emulation (RISE) or instruction-set randomization (ISR) [Barrantes05, Kc03].

The basic operation of an ISR system is as follows. An encryption algorithm (typically XOR'ing the instruction with a key) is applied statically to an application binary to encrypt the instructions. The encrypted application is executed by an augmented emulator (e.g., Valgrind [Nethercote04] or Bochs [Lawton96]. 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 in general depends on several key 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.

Our implementation of ISR using the Strata Virtual Machine improved upon the prior art in three important ways:

- We used a strong randomization algorithm—the Advanced Encryption Standard (AES).
- We demonstrated that ISR using AES could be implemented practically and efficiently without requiring special hardware support.
- Our approach detected malicious code before its execution. Previous approaches had relied on probabilistic arguments that execution of non-randomized foreign code would eventually cause a fault or runtime exception.

2.4 Calling Sequence Diversity

While code-injections attacks constitute the overwhelming majority of attacks today, other forms of attacks exist that do not require the execution of foreign exploit code. For example, in a return-to-libc attack, an attacker supplies malicious arguments to existing library functions with disastrous consequences. For example, supplying "bin/sh" to the system() function will execute a shell and provide an attacker with full-featured access to the target host.

The typical return-to-libc exploit is possible because an attacker is able to disrupt the intended control flow of the target program through manipulation of the return address (often through a buffer overflow vulnerability).

Note that such an attack may be thwarted by the *Address Space Randomization* or *Stack Space Randomization* techniques.

However, this style of attack critically depends on the attacker's knowledge of the calling convention. *Calling Sequence Diversity* provides a secure calling convention that prevents unauthorized invocation of potentially malicious functions. Our approach to developing such a

calling convent was to require a hidden parameter that was checked by the called function. Since attackers do not know the value of this parameter, they cannot execute the function successfully.

Strata was used to automatically and dynamically insert and check this random key to thwart return-to-libc attacks. For more details, refer to Appendix B, which incorporates a writeup of this technique.

2.5 Genesis Diversity Toolkit (GDT) Evaluation

This section presents an overview of the security and performance evaluation of the Genesis toolkit.

2.5.1 Security Evaluation

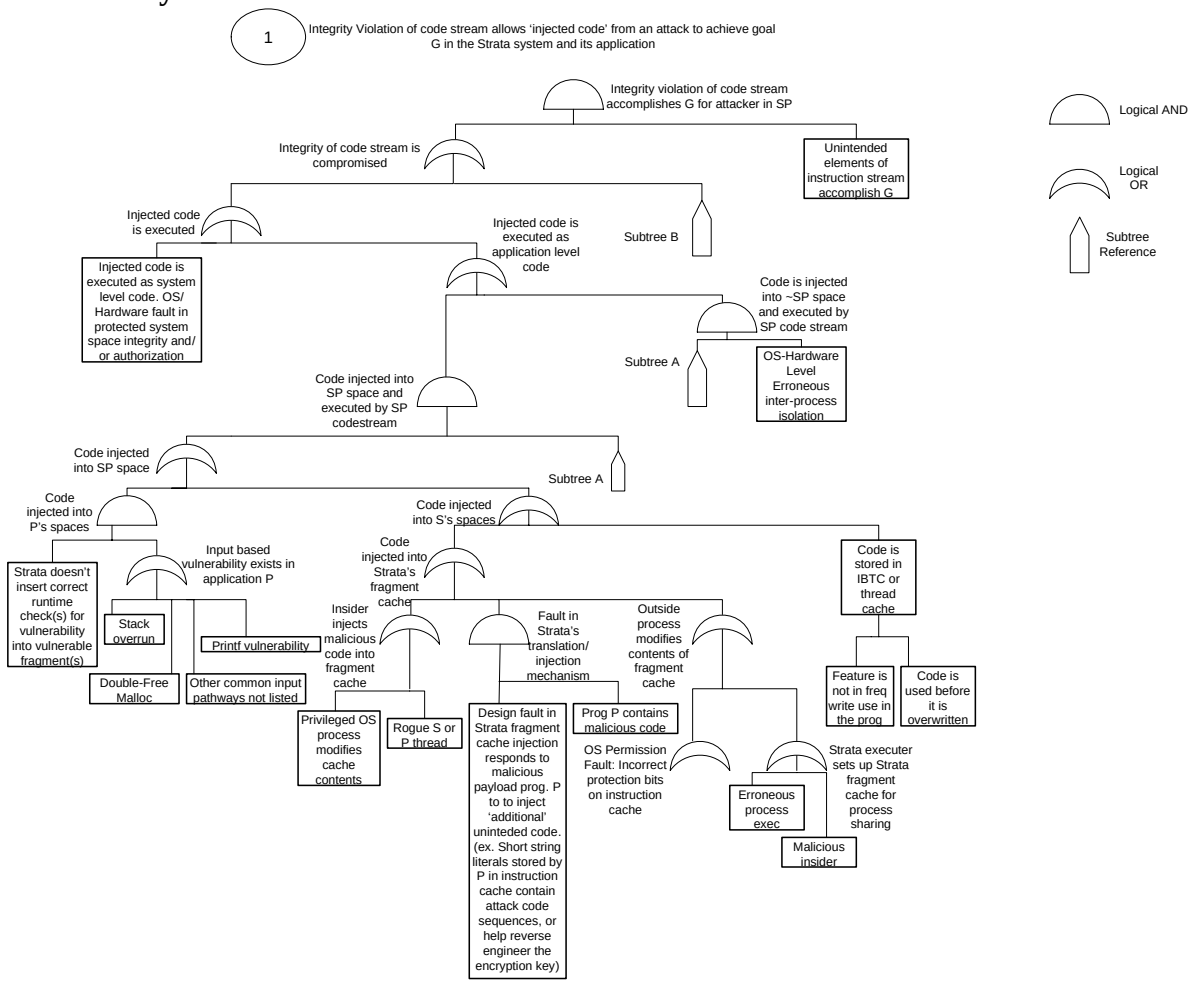


Figure 3. Sample Genesis Fault Tree

To analyze and demonstrate the strength and soundness of Genesis, we performed several experiments in which we ran applications with known-vulnerabilities under control of Genesis. We then ran the associated exploits on hundreds of variants generated by the GDT. Example vulnerabilities included buffer overflows and format string vulnerabilities targeted towards both the heap and stack. The success rate for ISR (for code-injection attacks) and for Calling Sequence Diversity (for return-to-libc

attacks) were both 100%. We also seeded applications with our own vulnerabilities, developed associated attacks and achieved the same success rate.

However, these experiments represented only point samples in the space of attack implementations. To argue for the soundness and broad applicability of our techniques, we developed a fault-tree to enable a comprehensive analysis of the Genesis design and implementation. This process uncovered a few omissions that we fixed and fed back into the implementation of the system.

The top level of the fault tree is shown in Figure 3. The top node identifies the goal state, namely that attack code was successfully injected and executed by the Strata virtual machine. The tree refines this hazardous states using AND and OR gates and details the necessary conditions required to reach this goal state. By systematically identifying goals and subgoals, and by stating any required assumptions, the fault tree provides a formal method of communications by which to evaluate the system design.

In addition to our own evaluation, we participated in two independent red team exercises commissioned by our DARPA program manager. The first red team exercise evaluated the strength of the GDT using the threat model of a remote attacker. We provided the red team with vulnerable applications and associated exploits which they used as a starting point for the exercise. The red team was unable to exploit applications running under the GDT. However, due to resource constraints, this exercise was limited in scope.

Thus, we undertook a second red team evaluation in which the scope of the exercise was expanded dramatically to encompass not just applications but also the virtual machine used as part of the GDT. In addition we provided the red team with our fault tree and associated analyses (Figure 3, Appendix C). The main idea behind this second exercise was to emulate a sophisticated insider as an attacker, *i.e., what if one of the developer of the system was part of the attacking team?*

Overall, this second exercise resulted in a blue team (Genesis) victory and validated the basic soundness of our design and implementation. However, we note that both exercises were limited in scope and duration. A more thorough red team evaluation would be needed prior to large-scale deployments on a DoD system. Reports for both red team exercises are available through the program manager.

2.5.2 Performance Evaluation

Overall, the implementation of the *Instruction Set Randomization* and *Calling Sequence Diver*sity techniques did not add much overhead to the baseline case of running applications under the Strata Virtual Machine. At first glance, this result may appear counter-intuitive in light of the fact that ISR uses the AES algorithm and that other implementations of ISR used a very simple XOR encoding scheme because of performance consideration [Kc03, Barrantes05]. The use of aggressive caching techniques in the Strata virtual machine enabled efficient implementations of diversity transformations since even relatively heavyweight operations such as AES decryption were only performed once for a given code fragment throughout the execution of a program. Thus as the working set of the program materialized in Strata's code fragment cache, little additional overhead is incurred as the program code executed natively.

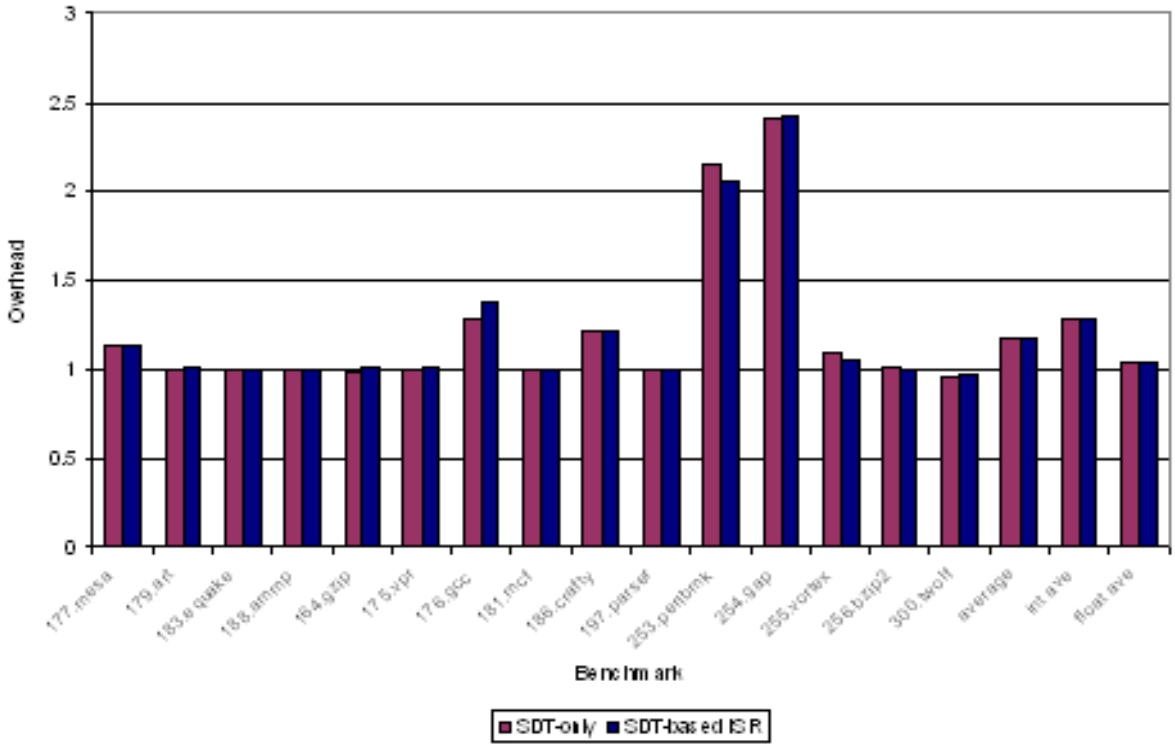


Figure 4. Strata and Strata+ISR Overhead Normalized to Native Execution (SPEC)

Thus the incorporation of diversity techniques and other security measures were to a first-order approximation essentially free: *the efficiency of diversity transformation techniques essentially depends on the efficiency of the Strata virtual machine*. As shown in Figure 4 for the SPEC benchmark, there was little difference between the two scenarios.

The implication is that as virtual machine technology continues to mature and becomes more efficient, diversity-based techniques will become even more practical. For example, during the period of this award, Strata performance improved from 30-40% average overhead to under 10% for applications such as Apache and BIND.

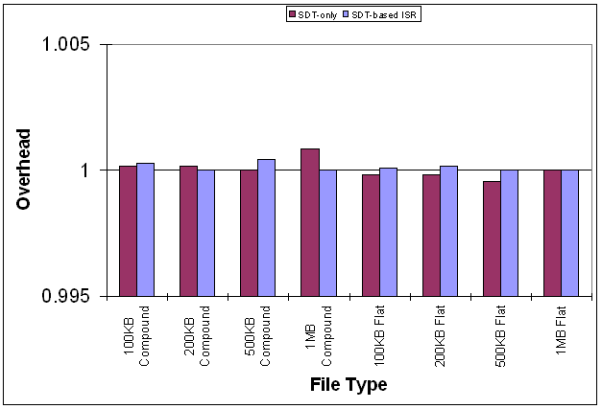


Figure 5. Apache Overhead Normalized to Native Execution. Metric: client request/sec

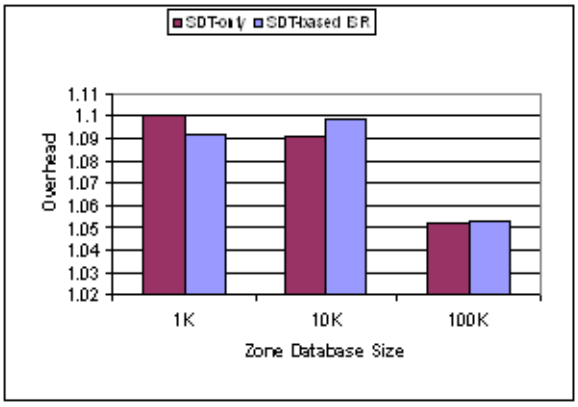
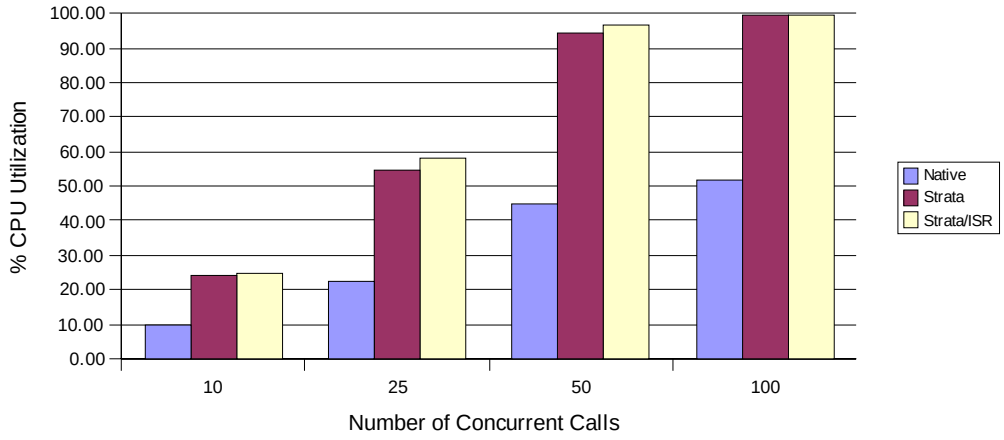


Figure 6. Bind Overhead Normalized to Native Execution. Metric: queries/sec

In terms of absolute performance relative to native execution, we achieved average overhead of 17% on the SPEC benchmark (Figure 4), less than 5% running the Apache web server (Figure 5), and between 5-11% on the BIND server (Figure 6).

We also measured the overhead on a commercially available GSM software-defined radio (SDR) package. The primary metric was a Quality-of-Service (QoS) measurements of the ability to sustain 100 concurrent call sessions for a protocol converter application within the SDR package. The Strata-based version saturated the CPU at 100 concurrent calls. After 100 calls, there could be a loss of fidelity because less signal data is being communicated per channel. In discussion with the engineers of the system, 100 calls was well within the acceptable range of operation and in fact represented a high testing watermark (i.e., in actual deployments, they did not expect to reach this level).



On a micro-benchmark stress test we achieved 85% overhead. While this result seems high, we note that this was a stress test that ran at faster than real-time in the sense that the processing speed was higher than the input sampling rate of the audio stream. We would need to measure macro-benchmark application-level overheads to determine the real-world performance implication of our approach.

The use of Strata (and in fact, of virtual machine technology in general) in soft-real time applications is a relatively uncharted area. Our preliminary results gave us confidence that with further performance optimizations, we could reduce the CPU load and performance overheads further.

2.6 Genesis Toolkit Enhancements

In the last phase of the project we worked towards improving the maturity and stability of the Genesis toolkit. This was necessary to pave the way for an eventual tech transfer to potential partners and for enabling the evaluation of the toolkit on real-world commercial applications. Improvements made to the GDT are described below:

- **Support for the Debian GNU/Linux operating system**: Genesis was developed on a variety of standard UNIX platforms, but had not been tested under this environment. Minimal work was required to port the Genesis toolkit to Debian as previous work had been tested under Red Hat Linux.
- **Robust support for C++**: The Genesis toolkit was originally implemented to support the C programming language. Because the C and C++ run-time

initialization takes place before the main() function is entered, we moved the transfer of control to Strata to the very beginning of the process execution by effectively prepending a few Assembly call instructions to the executable. This put the global constructors and destructors under the control of Strata. Not a single application instruction is executed outside the control of the Strata VM as of this change.

- **Robust support for multi-threaded software**: While Strata did already support threaded applications, a number of previously esoteric bugs were brought to light. Running the pthreads unit tests for the GNU libc implementation no longer breaks any previously working test cases.
- **Improved engineering procedures and documentation**: A large number of tests were added to the engineering process. A number of third party tests have been adopted, e.g. GNU libc unit tests, Linux threads unit tests, gzip package self tests.
- **Independence from the C library**: To support immediate control of the hosted application by Strata, we needed to remove all dependencies upon code in the standard C library. This allowed us to stop linking in a second copy of libc to avoid sharing code at runtime between Strata and the hosted application. A total of 48 functions were reimplemented in a stand-alone library. Of these only 18 were OS system calls. As a side effect of this change, the increase in the application size as a result of embedding it in Strata was reduced by 44%.
- **New logging infrastructure**: A great deal of refactoring was done to create a consistent logging facility. This was necessary to reduce the dependencies upon the C library before implementing the stand-alone library replacements. Logging was split into two parts: Strata logs binary data to a persistent store and a log viewing client is used to render the log messages into human readable text.
- **Build Process Integration**: We modified the default configuration of the gcc compiler such that a minimal number of options, often zero, were needed. As a result the GDT tool chain can now be used to compile applications with little modification of the sources.

Additional information about this phase of the project is available upon request. It was not included in this report because of confidentiality agreements.

2.7 Recommended Configuration

The recommended configuration for the Genesis Diversity Toolkit depends on the threat model assumed. Since code-injection attacks still predominate today, and will likely continue do to so for the foreseeable future, we recommend the use of ISR to protect networked applications. If more security is desired then all the other diversity transformations of the Genesis toolkit could be combined.

If the threat model is that of exploitable vulnerabilities coupled with concerns about code integrity, unauthorized copying and execution of applications, or reverse engineering, then we recommend the static version of ISR in which binaries are statically encrypted and decrypted only as the program executes. However, this configuration option has wide ramifications for the deployment and maintenance lifecycle of an application. Instead of producing a single image of an application and distributing identical copies, this option would require the production at the factory of *n* program variants, each with its own randomization key.

3 Summary of Results

In this section, we present a summary of the results obtained by the Genesis team.

3.1 Security Benefits of Genesis

Using the Genesis Diversity Toolkit (GDT), C and C++ applications were protected against code-injection and several forms of arc-injection attacks, including return-to-libc style attacks. Code-injection attacks represented (and still represent) a large and important attack class that afflicted popular Internet-enabled software applications such as Web, database, and domain name resolution servers.

Unlike previous diversity-based approaches, the GDT provided techniques that: (1) protected against all code-injection attacks regardless of the exploit path through a program, (2) did not rely on probabilistic guarantees for attack detection, and (3) used strong cryptographic protocols such as the Advanced Encryption Standard (AES).

Instead of targeting specific types of vulnerabilities, e.g., format strings, buffer overflows etc…, the GDT provided broad protection against both known and unknown code-injection attacks. In addition, the GDT also provided protection against return-to-libc attacks, though these were and still are not nearly as prevalent as code-injection attacks.

The GDT was evaluated by two independent red teams. These exercises were conducted under relatively tight budgetary and time constraints. Thus, while both exercises validated the soundness of our approach—the red teams were unable to penetrate our defenses or find major flaws with our system design—more thorough investigation would be required prior to large-scale deployment.

3.2 Genesis Diversity Toolkit Status

The GDT supported C and C++ programs running on the Red hat Linux 7.x and Fedora Core, and Debian 3.x operating systems. The GDT was evaluated on several real-world and relatively large applications, including the Apache web server, the BIND name resolution server, and a proprietary software-defined radio application, in addition to SPEC benchmarks. We achieved average overheads of 17% on the SPEC2000 benchmark, 5%-10% on BIND, less than 5% on Apache. For a commercial soft real-time application, we were able to meet QoS requirements though at an increased CPU level. Preliminary performance data on a micro-benchmark stress-test yielded overhead of 85%. Further research and optimizations would be warranted for soft-real time applications in general, though early results were encouraging.

3.3 Patent Applications

The University of Virginia Patent Office filed for a patent application to cover the technological foundation underlying the Genesis toolkit in July 2006. This technology was licensed by a subsidiary of UTEK Corporation, a specialty finance company focused on technology transfer (http://www.utekcorp.com/).

We note that the patent application was broader than just protection against code-injection attacks and covered the execution of encrypted and encoded applications using virtual machine technology. As discussed in Section 5.1 and in Appendix D, the technology developed has applications to the field of anti-tampering, which is particularly well-suited to DoD military systems.

3.4 Technology Transfer

In addition to the license agreement described above, we have ongoing relationships with several companies, including large defense contractors and smaller players in the defense arena, to explore venues for continuing the development of the Genesis toolkit and for transitioning the technology to wider use. The GDT has reached a maturity level that makes it practical to evaluate its effectiveness on real-world military applications.

3.5 Other Results

The Genesis project has led to the establishment of several related research projects: *PhPrevent* and *Secretless Structures for Security (SSS)*. The former provided protection for web applications against such attacks as SQL injection, Cross-site scripting (XSS), and Script injections attacks. The latter sought to provide provable security guarantees against classes of attacks and was a direct extension of the diversity work funded by this project. Its notable feature was that by judiciously using diversity techniques, security could be provided without relying on any secrets such as randomization or cryptographic keys. This project is now funded by the National Science Foundation under the CyberTrust program.

4 List of Major Publications

The following list itemizes the major publications during the contract period, including a patent application that covered the execution of encrypted binaries using a virtual machine approach:

Method for Software Protection Using Binary Encoding. Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W. Davidson, David Evans,John C. Knight, Anh Nguyen-Tuong, Jonathan Rowanhill, Adrian Filipi. *Patent Application filed by the University of Virginia Patent Foundation in July 2006.* [Proprietary information not included in report. Available upon request by DoD personnel].

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. *2nd Virtual Execution Environments Conference*, Ottawa, Canada, June 2006. [Appendix A: Instruction Set Randomization].

Where's the FEEB?: The Effectiveness of Instruction Set Randomization. Ana Nora Sovarel, David Evans and Nathanael Paul. *14th USENIX Security Symposium*. Baltimore, MD. August 2005. [Appendix G: Derandomizing Attacks].

Automatically Hardening Web Applications Using Precise Tainting. Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, David Evans. *Twentieth IFIP International Information Security Conference* (SEC 2005). 30 May - 1 June 2005, Chiba, Japan. [Appendix F: PHPrevent – Web Application Security].

4.1 Website

Presentations, papers and summaries are available through our various web sites. An overview of Genesis is available at http://www.cs.virginia.edu/genesis; an overview of the web application security project is available at: http://www.phprevent.org; an overview of the Secretless Security project is available at http://www.nvariant.org; and finally an overview of related projects can be found at http://dependability.cs.virginia.edu.

5 Technology Transfer & Future Opportunities

Large-scale experimental evaluation and validation of the Genesis toolkit on actual or next-generation DoD systems represent our logical next steps given the maturity of the Genesis Diversity Toolkit. We have discussed this possibility with the Navy, with several companies and with a large defense contractor. So far, we have not secured commitments to undertake an expanded evaluation project, but are continuing discussions with interested parties.

Through the University of Virginia Patent Foundation, we have licensed the technology described in our patent application to a subsidiary of Utek Corporation. The UVA Patent Foundation is actively seeking further licensees.

5.1 Anti-Tampering Applications

Anti-Tampering is an area of high interest to DoD, especially for systems that are susceptible to capture by enemy forces or systems used by allies for which DoD seeks to protect its intellectual property. In Appendix D: Tamper Proofing, we provide a summary of Anti-Tampering using the Genesis toolkit, for providing protection against the unauthorized modification of software, against reverse engineering of software, and against piracy, i.e., running software on a different target host than was intended.

5.2 Recovery

Most diversity-based defense techniques are only able to detect and stop attacks. While this is a significant first-step in building self-regenerative systems, further research is needed in building software systems that can self-heal when faced with attacks.

A primary and critical characteristic of our Strata-based implementation of ISR was that instead of crashing a process like other diversity-based techniques, we actually maintained control of the application when attack code was executed. Furthermore, we could identify the attack code *precisely*. Thus, while our default policy is to exit a program upon detection of an attack, the use of the Strata virtual machine opens the door for more sophisticated policies that can analyze the captured code and effect repair and recovery actions *in situ*. These repairs could prevent further similar attacks from affecting the running program, thereby potentially increasing the availability of mission-critical information systems.

5.3 Finer-grained Diversity

To reduce potential windows of vulnerability to a finite user-controlled time bound, we would like to investigate the use of our virtual machine technology to re-diversify applications dynamically (temporal diversity) while they are executing, without having to restart applications, and with little impact on performance. A simple and relatively easy to implement policy would be easy to dynamically change the AES key used in our ISR implementation. Preliminary results indicate that flushing the instruction fragment cache periodically on the order of seconds incurs acceptable overhead. Additional policies to obfuscate or diversity the application code would result in a program that presented a fast and dynamically shifting attack surface to attackers.

6 Conclusion

The Genesis team has developed novel techniques for defending against large classes of attacks, including return-to-libc and code-injection attacks, and incorporated these techniques into the Genesis Diversity Tookit (GDT). The GDT was developed to a maturity level sufficient for protecting real-world C and C++ applications. We have validated the soundness of our virtual-machine based approach and architecture through both experimental evaluation and red team exercises. We have demonstrated the practical benefits of our approach, with only minor performance overheads on several real-world applications of interest to the DoD. We have also demonstrated the GDT on soft real-time applications, using software-defined radio as our target application. We feel that the GDT is now ready for evaluation on further DoD systems. Furthermore, we have identified several promising areas for future research, notably in the field of anti-tampering for critical military systems.

7 References

[Nergal01] Nergal. The advanced return-into-libc exploits: PaX case study. Phrack Magazine 58, Article 0x04. 2001. http://www.phrack.org/phrack/58/p58-0x04

[Barrantes05] 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.

[Kc03] 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.

[Lawton96] Lawton, K. P. Bochs: A portable pc emulator for unix/x. Linux J. 1996, 29es (1996), 7.

[Nethercote04] Nethercote, N. Dynamic binary analysis and instrumentation. Tech. Rep. UCAM-CL-TR-606, University of Cambridge, Computer Laboratory, Nov. 2004.

Appendix A: Instruction Set Randomization

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

By Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W. Davidson, David Evans,John C. Knight, Anh Nguyen-Tuong, Jonathan Rowanhill. Published in *2nd Virtual Execution Environments Conference*, Ottawa, Canada, June 2006.

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 randomization algorithm—the Advanced Encryption Standard (AES). AES is generally believed to be imprevious 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.

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 buffer overflow vulnerability is a *code-injection* attack. In a code-injection attack, an attacker exploits the buffer overflow vulnerability 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 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 statically 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 key 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 [24]. 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 before it is readied for execution.

Extensive testing of our approach (including an attack exercise carried out by third-party security experts on an *Apache* web server that had been seeded with vulnerabilities) revealed no security breaches.

The implementation is practical. Rather than use emulation or postulating hardware extensions, we use a robust, efficient software dynamic translation (SDT) system [21]. Performance measurements using a variety of benchmarks including the full SPEC CPU2000 suite, a domain name server, and a web server, showed the runtime overhead of SDT-based ISR to be modest—16% for SPEC CPU2000, 6–10% for the domain server, and no overhead for the web server. Space overhead of SDT-based ISR is also reasonable—the text size of a protected web server was 53% larger than an unprotected web server. However, the working set size of the two implementations were similar. More detailed measurements of the overheads of ISR are reported in **Performance Evaluation**.

The remainder of this paper is organized as follows. **Threat Model** describes the class of attacks that ISR handles. Previous work on ISR is described in **Previous Work**. **Secure and Practical ISR** describes our SDT-based implementation of ISR. An evaluation of the security and performance of our approach is given in **Evaluation**. **Related Work** gives an overview of related work, and **Summary** 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 [19, 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, there are several that are not covered. The model does not cover arc-injection attacks (also known as return-to-libc) [18], 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 independently developed by groups at the University of New Mexico [3] and Columbia University [9]. Both groups implemented ISR prototypes for the x86 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 can be cracked with modest effort [23]. It is also important to note that the use of a one-time pad the length of program effectively doubles the program size. For some applications, a doubling of code size could be problematic.

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.

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 [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. 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. Binary rewriting is used to prepare the binary for strong encryption and introduce the information necessary to detect foreign code before it is executed.

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. 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 [20], profiling [12], and code compression [22]. 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 Strata virtual machine virtualizing an application.). 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, a *context switch* restores the application context and begins executing cached translated instructions on the host CPU.

If there is no cached translation for the next application instruction, the Strata VM allocates storage 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.

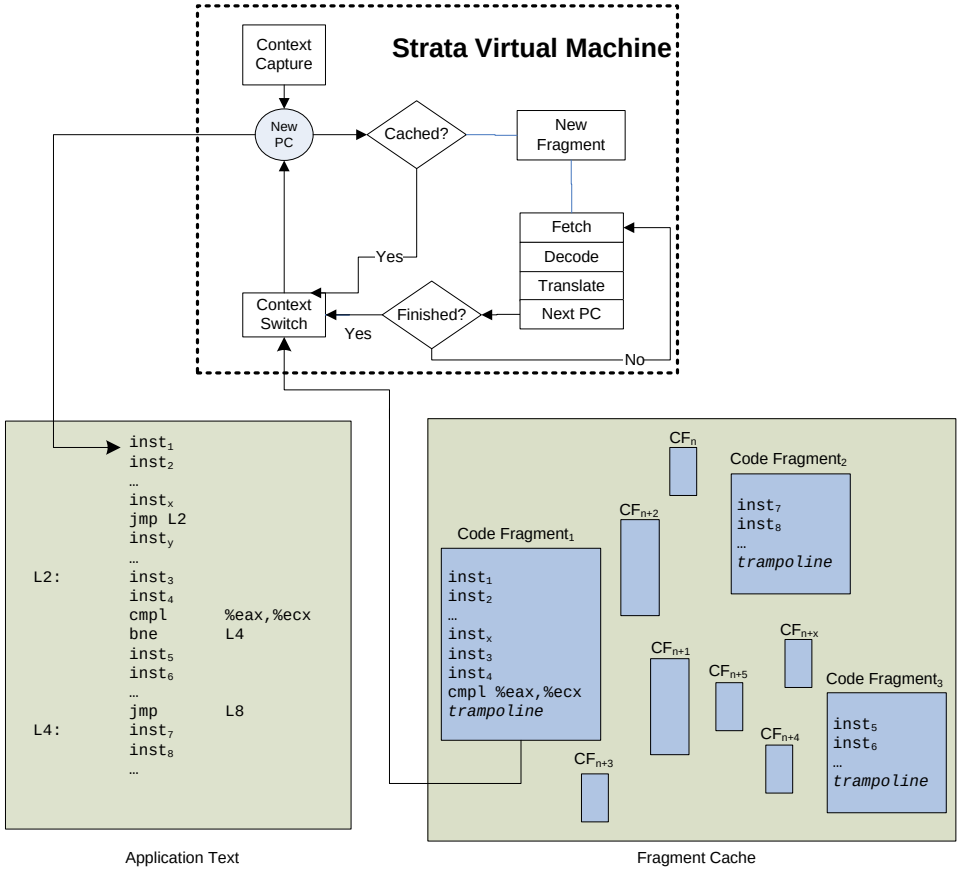


Figure 1: Strata virtual machine virtualizing an application.

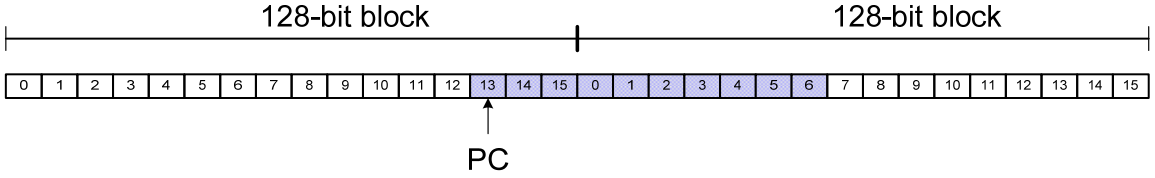
The implementation of ISR required two simple extensions to Strata. First, we introduced an encryption feature that applies AES to the application text before Strata begins execution of the application. Second, we overrode Strata's default fetch mechanism. The new fetch method decrypts and verifies an instruction before calling the default target-machine fetch method.

Runtime decryption and verification. gives the basic steps Strata carries out to implement ISR. In Step 1, Strata's security API is used to intercept all system calls to mprotect and sigaction. This is done to prevent 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.

Step 2 encrypts the binary. The binary rewriting process created and embedded in the application text a table, 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 is then write protected.

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 x86 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 **Binary Preprocessing**), each instruction is tagged with a simple eight-bit MAC (message authentication code). After decrypting the two blocks, Strata checks the MAC to ensure that the fetched bytes represent a valid application instruction. If the MAC is valid, Strata simply invokes the default fetch method with the PC pointing at the first byte of the instruction.

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 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. Write protect the text segment.
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.

The use of an eight-bit MAC means that there is a 1 in 256 chance that the MAC is coincidentally correct. However, in order for Strata to execute the fragment containing the injected code, the MAC for each instruction in the fragment must be correct. Thus for a fragment containing four instructions, the probabilty that four individual MAC would be coincidently correct is 2^{-32}. However, with no penalty in runtime, the size of the MAC could easily be increased.

If the MAC 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 an invalid MAC is detected, Strata 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.

It is important to note that the process of decrypting the application text, checking the MAC, 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. **Performance Evaluation** provides detailed measurements of the overheads of our SDT-based implementation of ISR.

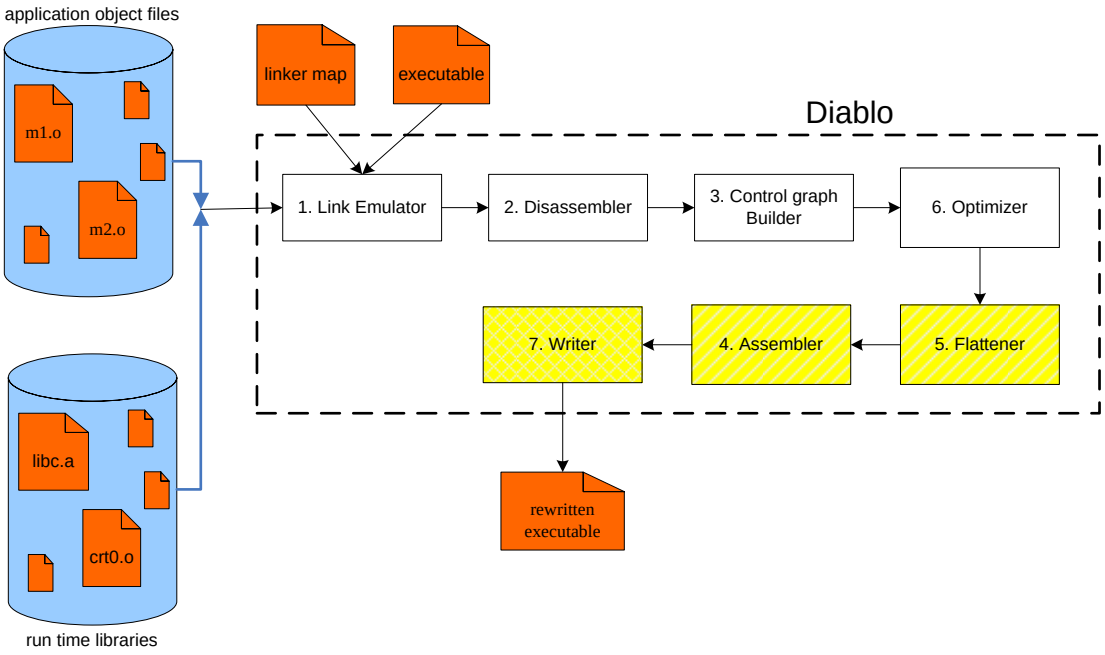


Figure 3: Work flow of the binary rewriter Diablo.

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 back into 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.

Our current implementation of ISR 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 employed self-modifying code. Nonetheless, we plan to investigate techniques for safely handling legitimate uses of self-modifying code.

4.3 Binary Preprocessing

To prepare the binary for encryption using AES and to introduce the necessary MACs, we modified Diablo, an existing binary rewriting tool [4]. Work flow of the binary rewriter Diablo. 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. In phase 2, Diablo dissects and translates the program into an internal representation. In phase 3, Diablo disassembles the instructions and 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 Work flow of the binary rewriter Diablo. 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 Diablo extensions to support ISR..

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, reserve one byte before each instruction for a MAC.
   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 basic block do:
   a. Determine the source of the basic block.
   b. If the basic block is application code and instruction tagging is enabled, insert a MAC in the each instruction in the block.
   c. If the basic block is application code and instruction tagging is disabled, insert an NOP in the space reserved for the MAC.
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 segment, the data segment, the encrypttable segment, and any other segments).

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 MAC before each instruction. It then aligns the block appropriately by padding the beginning of the block with NOPs (these NOPs are elided by Strata during its translation process).

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. After all blocks are processed, the Flattener recalculates branch offsets and updates all instructions affected. The starting address and length 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 MAC if instruction tagging is enabled, or with a NOP if instruction tagging is disabled. Again, Strata's translation process elides NOPs.

The final phase of Diablo emits the modified binary to disk. This phase of Diablo was extended to create a new segment, .encrypttable, to contain the encryption information that Strata uses to initially encrypt the application text.

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.

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 into several real applications and then exploited the vulnerability to effect a code-injection attack. Table I lists the applications, the type of vulnerability, and the target memory region of the injected code. For each vulnerability, we demonstrated that exploitation of the vulnerability could be used to compromise an unprotected system. In all cases, ISR was able to detect the attempt to execute injected code and prevent the attack from proceeding.

Our protected version of *Apache* was also subjected to attack by a security team consisting of several security experts. All algorithms, source code, and design documents were provided to the security team for analysis in advance of the exercise. The target system was seeded with several known vulnerabilities and subjected to concerted code-injection attacks. The system was able to stop all attacks.

Application	Vulnerability	Location of injected code
Apache	Buffer overflow	Stack
Apache	Format string	Heap/Stack
Samba	Buffer overflow	Stack
Bind	Format tring	Heap/Stack
rpc.statd	Format string	Global offset table
cvs server	double free	Stack

Table 1: Tested applications.

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 reported in this section were taken on a 2.8GHz P4 Xeon with 512MB of RAM. Hyperthreading was enabled. The installed operating system was RedHat 8.0. In the case of the client/server applications, *Apache* and *Bind*, client processes were run on separate, but identical machines.

SDT overhead and SDT-ISR overhead normalized to native execution. Metric: SPEC ratio. shows the performance results for SPEC CPU2000. We measured the overhead of the baseline SDT system (no ISR), and the overhead of the SDT-based ISR system. The performance metric used to compute the overhead was the reportable SPEC ratio produced by the SPEC measurement infrastructure.

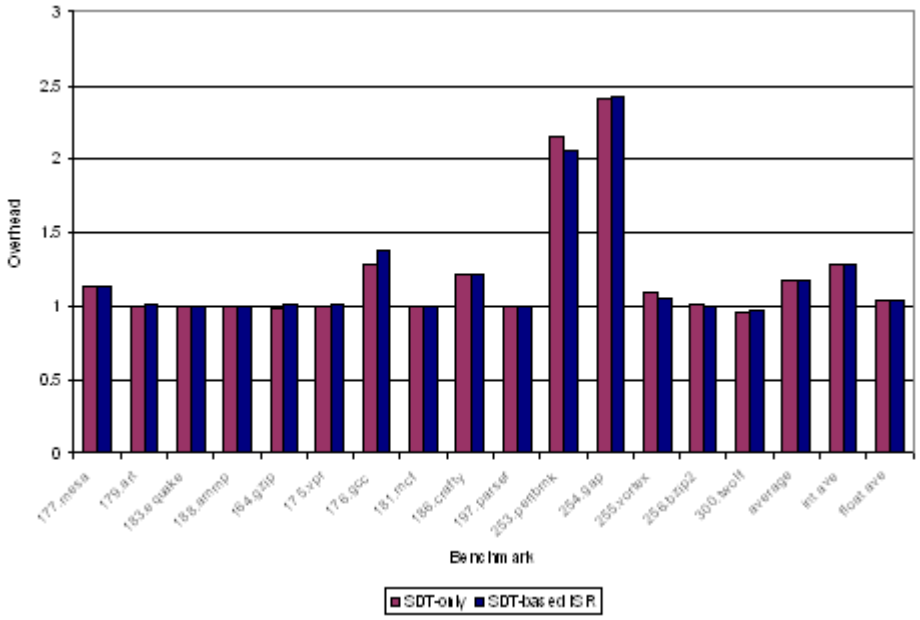


Figure 5: SDT overhead and SDT-ISR overhead normalized to native execution. Metric: SPEC ratio.

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, 21].

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 *flood*, the web server performance measurement tool developed and supported by the Apache Software Foundation. *Flood*'s performance metric is number of client requests served per second. For the measurements reported here, *flood* was configured to spawn eight clients each requesting pages from the server. We confirmed that the server was saturated with requests.

To determine if performance was sensitive to the size of the page served, we measured performance using a variety of page sizes. We also measured the performance when the web page served consisted of several components. In these instances, the sum of the components is reported. For example, the label "200KB Compound" indicates that the web page served consisted of several individual files and the sum of the sizes of the individual files was 200KB. *Apach* contains the results.

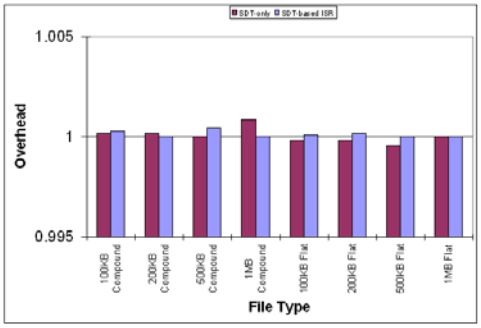


Figure 6: *Apache* overhead normalized to native execution. Metric: client requests served per second.

As the chart shows, the overhead of running either the baseline SDT system or the SDT-ISR system is negligible. The performance is independent of the size of the web page being served or whether it is a single, flat file or a page consisting of several individual components.

To measure the performance of the Berkeley Domain Name server, *Bind*, we created three representative zone files. Briefly, a zone file contains directory 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 measurement results are presented in *Bin*. 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-ISR system.

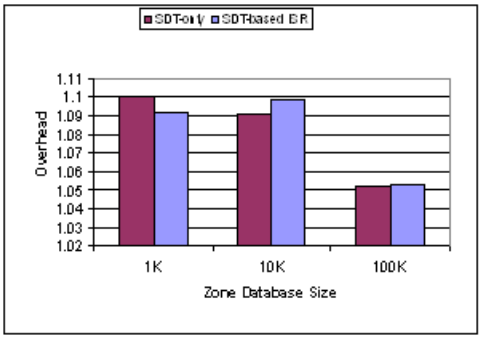


Figure 7: *Bind* overhead normalized to native execution. Metric: queries per second.

We also measured the space overhead of SDT-based ISR. For *Apache* and *Bind*, the text size of the rewritten binary was 53 and 57 percent larger, respectively, than the text size of the corresponding native binaries. Most of this overhead was due to the one byte MAC inserted before each instruction.

The encrypttable which was stored in a separate segment increased the size of the initialized data segments by 16% for *Apache* and 40% for *Bind*. The size of the encrypttable depends on the number of encryption blocks. For all performance experiments, the size of Strata's fragment cache was fixed at 4MB.

We also observed the working set size of the running applications. We saw no difference in the working set size between the SDT versions and the native versions. This is not surprising as Strata itself is small and only the code that is executed materializes in the fragment cache.

While we believe that the size overheads are reasonable for server applications, for some environments reducing space overhead may be desirable. The size of the text segments could be substantially reduced by computing a MAC for a block of instructions rather than a one-byte MAC for every instruction.

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 [25] are two popular example of such methods.

Previous work involving software-base implementations of ISR is described in **Previous Work**. 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 [20]. 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.

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. Similarly, 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.

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. In addition, an *Apache* server was seeded with vulnerabilities. The server along with detailed information about the seeded vulnerabilites was delivered to a set of security experts for analysis and testing. The seeded server was subsequently subjected to a concerted set of attacks by the security experts. The ISR-protected web server was not compromised.

The SDT-based implementation of ISR is sufficiently efficient to be used to protect critical service applications that are often the target of attack. Measurements of an ISR-protected *Apache* web server showed little or no performance loss 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. [20]

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. 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 exttrademark: 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* (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] PINCUS, J., AND BAKER, B. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security & Privacy 2*, 4 (jul/aug 2004), 20–27.

[19] 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.

[20] SCOTT, K., AND DAVIDSON, J. Safe virtual execution using software dynamic translation. In *Annual Computer Security Applications Conference* (2002).

[21] 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.

[22] SHOGAN, S., AND CHILDERS, B. Compact binaries with code compression in a software dynamic translator. In *Design Automation and Test in Europe* (2004).

[23] 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).

[24] THE COMMITTEE ON NATIONAL SECURITY SYSTEMS. National policy on the use of the advanced encryption standard (aes) to protect national security systems and national security information.

[25] THE PAX TEAM. http://pax.grsecurity.net.

Appendix B: Calling Sequence Diversity

Calling Sequence Diversity

1. Introduction

Software vulnerabilities manifest themselves in many different ways. The traditional stack smashing attack takes advantage of a buffer overflow vulnerability to inject malicious code on to the stack and execute it [1]. In order to protect against such an attack, several defenses that effectively render the stack non-executable have been proposed. While this approach will certainly thwart a stack smashing attack, it is not foolproof.

A return-to-libc attack can bypass a non-executable stack by relying on existing code. Instead of executing injected exploit code, the attack uses existing library functions that can have malicious consequences with certain arguments. A simple example involves pointing the return address to the `system()` library function with the argument pointing to an instance of the string `"/bin/sh"`. When the compromised function returns, a shell is executed without ever using any injected code [2].

This paper proposes a method that will detect and prevent return-to-libc attacks and other attacks that take advantage of existing code. It is intended as a supplement to existing schemes that already thwart code injection attacks.

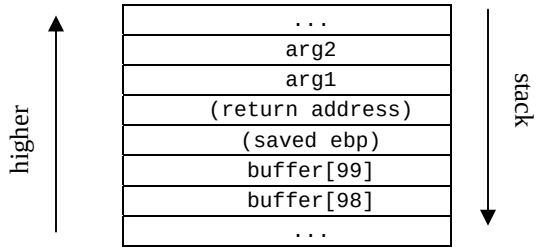
2. Return-to-libc Exploits

In order to understand how a return-to-libc attack works, consider the vulnerable function `foo()` shown in Figure 1a. The contents of the stack during the execution of `foo()` are shown in Figure 1b.

```
void foo(int arg1, int arg2)
{
    char buffer[100];

    ...
    scanf("%s", buffer);
    ...
}
```

(a) Vulnerable function



(b) Contents of the stack

Figure 1

A buffer overflow vulnerability is present in `foo()` because `scanf()` will overwrite data beyond the allocated memory for `buffer` if the user provides a string that is longer than 100 bytes. Looking at the stack, we can see that this data will overwrite the return address for `foo()`. Since the attacker can overwrite the return address for `foo()`, she can redirect control to any location she chooses once `foo()` returns. In the case of a return-to-libc attack, the return address will be changed to the address of a library function.

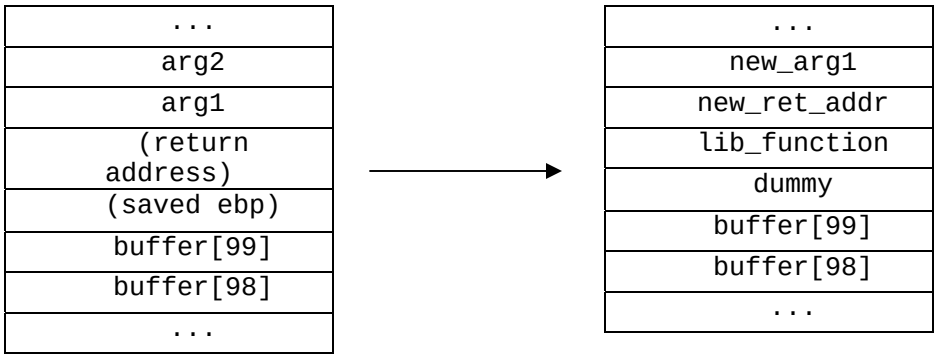


Figure 2: Overflowing the stack

Figure 2 illustrates how to overwrite the stack in order to cause a return-to-libc attack. The saved base pointer is overwritten by the arbitrary `dummy` value, since its value is unimportant. The return address is replaced by `lib_function`, the address of the library function to be executed. The value `new_ret_addr` will act as the return address for `lib_function`, as if it were pushed onto the stack during a normal function call. Following `new_ret_addr` is the argument list for the call to `lib_function`, starting at `new_arg1` [2].

Consider the case where the attacker wants to execute the `system()` function, which takes one string argument that is executed as if it were entered as a command in a shell. If the attacker sets `lib_function` as the address of `system()` and sets `new_arg1` as a pointer to the string `"/bin/sh"`, this will execute a shell whenever `foo` returns.

3. Approach

The return-to-libc exploit is possible because the attacker is able to disrupt the intended control flow of the program through manipulation of the return address. In order to prevent this type of attack we need a technique to ensure that any execution of library code comes from a legitimate function call in the program.

One way to enforce legitimate execution of library functions is to develop a calling convention that prevents unauthorized invocation of potentially malicious functions. Our approach to developing such a secure calling convention is to require a hidden parameter that will be checked by the called function. This hidden parameter would be an arbitrary value acting as a key. Since the attacker does not know what the key is, she will not be able to execute the function successfully.

3.1 First Attempt

Consider a simple implementation of this key system. Vulnerable library functions such as `system()` are identified and rewritten to accept an additional parameter. Code to verify that the parameter is the correct key is also added to these functions. The modified library code is recompiled to produce a new protected shared library.

Any calls to these protected functions must be changed to include the key as well. The compiler is modified to add the key as the additional parameter. Programs must be compiled with the modified compiler and linked with the modified library in order to receive protection.

This implementation will prevent the attacker from returning directly to a protected function because he must supply the key. However, it is possible to indirectly execute such a

function by returning to a point in the code that contains a legitimate library function call. This call would set up the key correctly and the call would succeed.

```
void main() {
    ...
    foo();
    ...
L1: system(command, key);
    ...
}

void foo() {
    // buffer overflow in foo sets
    // return address to L1
}
```

Figure 3: Returning to a legitimate call site

Consider the example in Figure 3. Instead of changing `foo`'s return address to the address of `system()`, the attacker changes it to `L1`, the location of a legitimate call to `system()`. This call has been compiled to include the key parameter, so the call to `system()` will succeed. In this situation the attacker is forced to use the `command` argument already in place at the call site, making it difficult to apply a set of malicious commands. However, it could be possible to overwrite the data pointed to by `command` if it is on the stack. Also, if a call site is found that gets arguments from registers or from the stack, it may be possible to manipulate these data locations in a way that gains control of the arguments.

3.2 Improved Key System

In order to account for this vulnerability, consider a different approach. Rather than passing the key as an explicit parameter to the function, the key is kept in a global variable available to all functions. Each function in the program has its own unique key. As the program enters and exits functions, the global key variable changes accordingly.

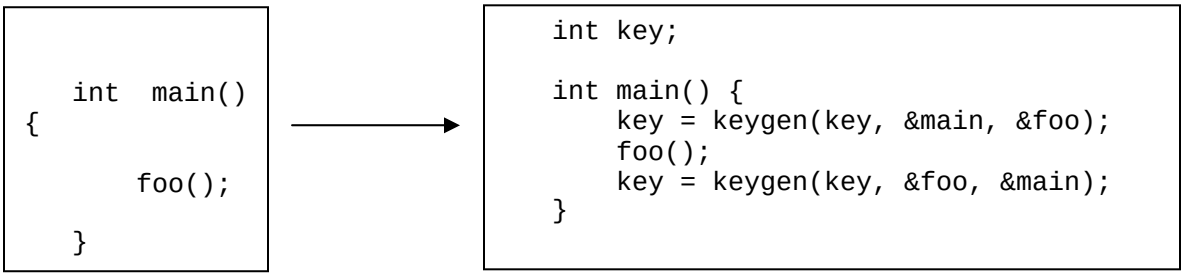


Figure 4: Key transformation

Figure 4 illustrates the basic idea behind key transformation. First, a global `key` variable is added to the program. It should be initialized to the key for `main()`. Whenever `main()` calls another function such as `foo()`, the key is transformed from `main`'s key into `foo`'s key through the `keygen()` process. When that function returns, the key is transformed back into `main`'s key.

Note that `keygen()` is not an actual function. It is merely pseudocode to illustrate the key transformation process.

The transformation is actually performed as an exclusive-or operation. Given a source function and a destination function, a constant is calculated. Performing an exclusive-or of this constant with the source key will produce the destination key. This is demonstrated in the following equation, where c is the constant and k_{src} and k_{dst} are keys for the source and destination functions respectively.

$$k_{src} \oplus c = k_{dst}$$

The transformation is achieved by providing a constant c that satisfies this equation. We can solve for c:

$$\begin{aligned} k_{src} \oplus c &= k_{dst} \\ k_{src} \oplus k_{src} \oplus c &= k_{src} \oplus k_{dst} \\ 0 \oplus c &= k_{src} \oplus k_{dst} \\ c &= k_{src} \oplus k_{dst} \end{aligned}$$

Therefore the constant c can be computed as an exclusive-or of the source and destination keys. As shown in Figure 4, the `keygen()` process uses three values: the value of the global key, and the addresses of the source and destination functions. The transformation is achieved by looking up the keys for the source and destination functions, performing an exclusive-or of the keys to produce the constant, and finally performing an exclusive-or of the global key with the constant. The resulting value is stored back into the global key variable.

This process may seem excessive, but there is an important property involved. The constant is calculated for specific source and destination functions, and therefore it assumes that the global key is already set to the key for the source function. However, if the global key is not set to the correct key, the transformation will produce a meaningless and incorrect value.

```
void main() {
    ...
    key = keygen(key, &main, &foo);
    foo();
    key = keygen(key, &foo, &main);
    ...
L2: key = keygen(key, &main, &system);
L1: system(command);
    key = keygen(key, &system, &main);
    ...
}

void foo() {
    // buffer overflow in foo rewrites
    // return address
}
```

Figure 5: Returning to a legitimate call site with key transformation

Consider this in the context of our example from Section 3.2 involving a jump to a legitimate call site. If the attacker modifies the return address to jump to some call in another part of the program, the global key will still have the value of the vulnerable function that he came from. This will be the wrong key for the transformation, and the resulting key will be wrong as well.

Figure 5 shows the example from Section 3.2 with the improved key system. The global `key` variable is changed to `foo`'s key before the call to `foo()`. Again, a buffer overflow in `foo()` overwrites the return address. However, if the attacker returns to the call at `L1`, the key will not be set to `system`'s key and the key check will fail. If the attacker returns to `L2`, the key transformation from `main()` to `foo()` will fail because the key is actually still set to `foo`'s key.

4. Implementation

If such a key system were implemented statically, the attacker could easily inspect the binary file and determine the value of the key. Therefore the proposed scheme will use a new random key every time the program is executed. First, the program needs to be compiled with the new calling convention. For this step I modified the Zephyr compiler infrastructure [3]. The key will be dynamically inserted into the code using Strata, a software dynamic translation tool [4].

4.1 Passing Information to Strata

A call to an actual `keygen()` function would incur significant overhead, so we would like Strata to insert the appropriate XOR instruction directly into the code fragment. Strata will need to know the source and destination functions in order to calculate the correct constant. If `keygen()` were compiled as a function call, it would be difficult for Strata to reliably determine the arguments. The compiler pushes arguments on to the stack, and may end up using registers or temporary variables in the process. Strata would need to backtrack through the code in order to determine the value of those arguments.

However, it is much easier to pass function addresses to Strata using function calls. Strata can determine the target of a function call without the need to look at any other code. We can accomplish this by using a sequence of three function calls. The first call to the reserved function `strata_key_direct()` acts as a placeholder to notify strata that a key should be transformed. Strata should then use the next two function calls to determine the source and destination functions for the transformation. When building a fragment, all three calls are discarded and replaced by the XOR instruction.

So, instead of the compiler generating our original code at a call to `foo()`:

```
key = keygen(key, &main, &foo);
foo();
key = keygen(key, &foo, &main);
```

The following code is generated:

```
strata_key_direct();
main();
foo();

foo();

strata_key_direct();
```

```
foo();
main();
```

The compiler produces the following assembly code:

```
call strata_key_direct  ; notify Strata of key transformation
call main               ; extract the address of main
call foo                ; extract the address of foo

call foo                ; original function call

call strata_key_direct  ; reverse the key transformation
call foo
call main
```

Finally, Strata will examine the binary and insert the following code into the fragment:

```
xor  %0x1234, key
call foo
xor  %0x1234, key
```

where 0x1234 represents the constant that is calculated by Strata.

However, this method will not work for indirect function calls since the destination of the function call is not constant. In this case we must incur the overhead of a call to a `strata_key_indirect()` function that will use the value of the function pointer when calculating the constant to use in the transformation. Figure 6 shows how a call through function pointer `fp` would be protected.

```
void main() {
    void (*fp)();
    fp = &foo;

    strata_key_indirect(&main, fp);
    (*fp)();
    strata_key_indirect(fp, &main);
}

void foo() {
}
```

Figure 6: Key transformation for an indirect function call

The `strata_keygen_indirect()` function will examine the values of its arguments in order to calculate the required constant. The global `key` variable will be transformed with an exclusive-or operation as usual. Since the `key` variable is global, it does not need to be passed as an argument. Strata will process the call to `strata_key_indirect()` as a normal function.

4.2 Compiler Modifications

The Zephyr compiler [3] is used to add the key transformation code at all function call sites. This was done by modifying lcc, one of Zephyr's possible front ends. The `keygen()` code is inserted after the intermediate language trees have been fully constructed by lcc. As these trees are added to the code list that will be passed to the code expander, they are checked for any function calls. If a function call is present, the `keygen()` code is inserted before and after the call.

4.2.1 Direct Calls

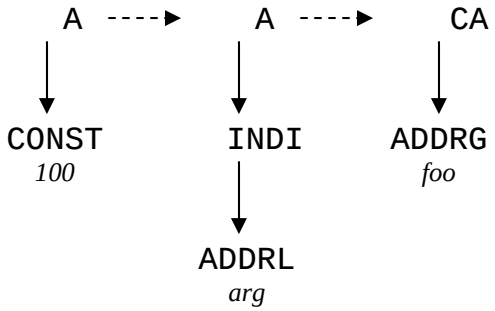


Figure 7: Intermediate language trees for `foo(arg, 100)`

As an example, Figure 7 illustrates the forest of trees that is constructed for the function call `foo(arg, 100)`. Solid arrows represent parent-child relationships within the trees, and dashed arrows represent links between trees in the forest. At this point in the compilation process, function arguments have been pulled out into separate trees, and the structure is quite simple.

The modified version of lcc traverses this forest, looking for a `CALL` node. If any such node is found, the appropriate `strata_key_direct()` calls are inserted before the first argument and after the original function call. Figure 8 illustrates the modification made to the forest shown in Figure 7, assuming that the function `foo()` is called from `main()`.

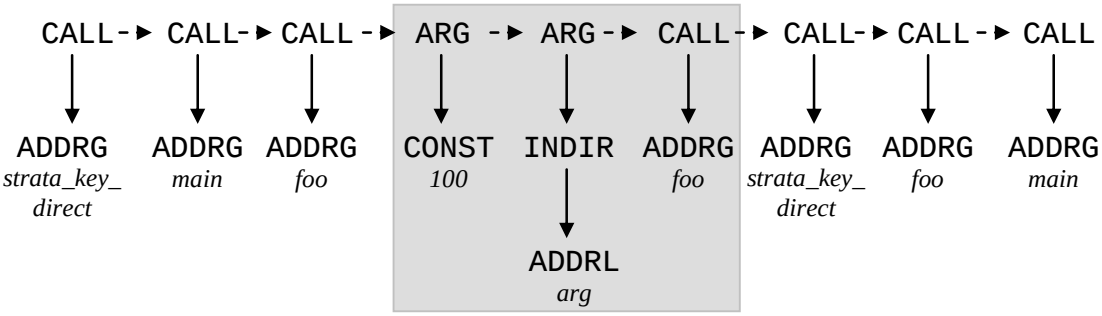


Figure 8: Modified intermediate language trees for `foo(arg, 100)` (*original call is shaded*)

4.2.2 Indirect Calls

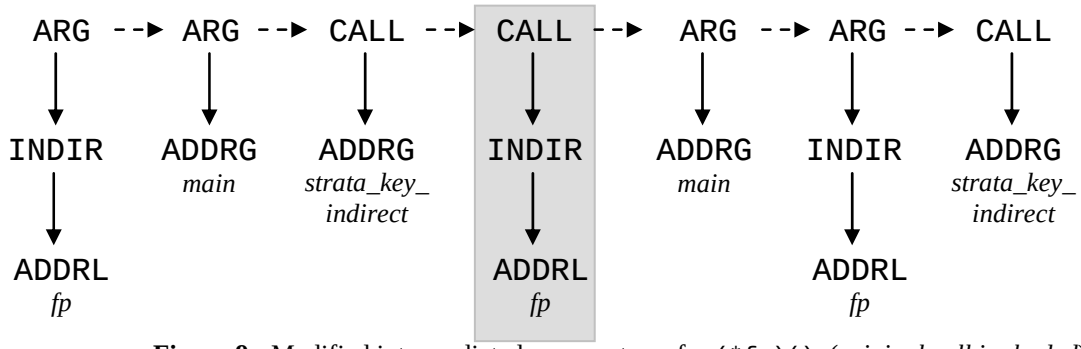


Figure 9: Modified intermediate language trees for `(*fp)()` (*original call is shaded*)

If a `CALL` node is found with an `INDIR` node as a child, the call is an indirect call and must be treated differently. A call to `strata_key_indirect()` must be inserted with the appropriate arguments. Figure 9 shows how to handle key transformation for an indirect call through the function pointer `fp`, again assuming that the call is made in `main()`. These modifications mirror the example shown in Figure 6.

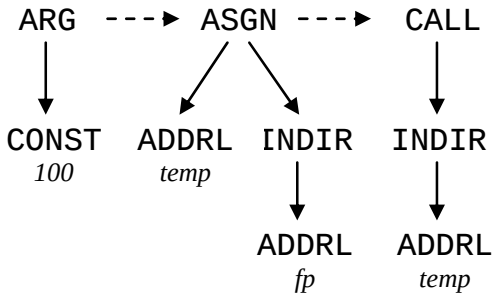


Figure 10: Problematic indirect call sequence

Unfortunately, certain compiler generated call sequences make the transformation in Figure 9 difficult. Figure 10 shows a sequence that requires multiple trees to calculate the address of the function to call. The call is performed through a temporary variable that is assigned in the previous tree. This is not a realistic example, but a simplification of much more complex call trees that use temporary variables to simplify the calculation of the function address.

The problem with the sequence in Figure 10 arises when generating the first call to `strata_key_indirect()`. We must place this call before the first argument tree, but the value of `temp` is invalid at this point because the assignment to `temp` has not taken place yet. The call to `strata_key_indirect()` cannot go after the assignment tree, because the compiler will assume that the existing argument should be associated with `strata_key_indirect()` rather than the original indirect call.

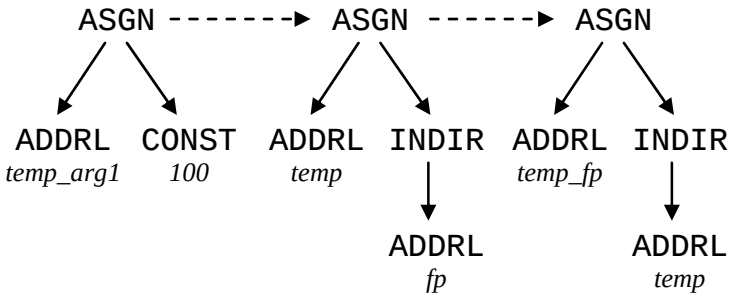


Figure 11: Saving call sequence values from Figure 10 for later

In order to avoid this problem, call sequence values are saved in new temporaries so that they can be used later. As the compiler advances through the code, it maintains a list of pointers to arguments that it has come across. If it turns out that these arguments belong to an indirect call, all arguments and the call itself are rewritten as assignments to temporaries. Figure 11 illustrates how the sequence from Figure 10 would be transformed. The argument is stored in `temp_arg1` and the function pointer is stored in `temp_fp` while the middle assignment tree remains unchanged.

Once these values have been stored in temporaries, the remaining work is simple. In addition to generating the calls to `strata_key_indirect()`, the original call must be reconstructed. All the values required are now stored in convenient temporary variables. The key transformation is achieved by generating the following code sequence (assuming the code is in `main()`):

```
strata_key_indirect(&main, temp_fp);
(*temp_fp)(temp_arg1);
strata_key_indirect(temp_fp, &main);
```

At this point, I will omit the tree structure for this code sequence, but it will be very similar to the sequence shown in Figure 9. Note that these new temporaries will generate some additional overhead, but they are only used in the event of an indirect call, which is already incurring more overhead than usual.

4.3 Strata Modifications

Strata is a software dynamic translation tool capable of modifying binary code on the fly [4]. Strata is used to dynamically generate and insert the key transformation code based on the placeholders created by the compiler. Strata also maintains the keys themselves, storing them in a hashtable for quick access.

4.3.1 Key Hashtable

The keys themselves are random 32-bit integers. The hashtable stores keys for each function and is indexed by the function address. It is structured as an array of linked lists. The index into this array is computed by selecting the least significant bits of the function address.

For ease of programming all access to the key hashtable is done through the `keytable_lookup()` function. This function takes an address as an argument. If there is a key installed for that address, its value is returned. However, if no key is found, a random key is

generated, installed, and its value is returned. Therefore the programmer can call `keytable_lookup()` whether or not a key exists for that address yet.

4.3.2 Call Translation

Our goal in modifying Strata is to recognize the calls set in place by the compiler and to replace them with the appropriate key transformation code. This goal is achieved by providing a custom `xlate_call()` function to the target interface. This function is responsible for translating any call statements encountered in the program text while building a fragment.

The custom function, called `xlate_key_call()`, examines the target address of all function calls. If a call to `strata_key_direct()` is found, Strata will fetch the next two instructions assuming that these are calls to the source and destination functions of the key transformation. A target address is computed for each of these function calls, and keys are retrieved from the hashtable for these addresses using `keytable_lookup()`. An exclusive-or of these keys is performed to produce the required constant. Finally, Strata emits code into the fragment to perform an exclusive-or of the global key variable with the constant.

In order to handle indirect function calls, the `strata_key_indirect()` function is implemented in Strata. The functionality is the same as the `xlate_key_call()` function described above with some obvious exceptions. The function addresses are supplied as arguments to the function, so these can be passed directly to `keytable_lookup()`. Also, instead of emitting code to perform the exclusive-or, it is performed as part of the function. There is no special code to handle calls to `strata_key_indirect()`. They are treated just as any other call.

4.3.3 `setjmp()` and `longjmp()`

The `setjmp()` and `longjmp()` functions in C are used transfer control across functions. When `setjmp()` is called, the stack environment is saved. When `longjmp()` is called later from another function, the stack environment is restored and program execution continues as if the corresponding call to `setjmp()` had just returned. This is a concern because the key will need to be maintained appropriately when jumping across functions.

```
void foo() {
    key = keygen(key, &foo, &setjmp);
    setjmp(env);
    key = keygen(key, &setjmp, &foo);
}

void bar() {
    key = keygen(key, &bar, &longjmp);
    longjmp(env);
    key = keygen(key, &longjmp, &bar);
}
```

Figure 12: Key transformation for `setjmp()` and `longjmp()`

Figure 12 illustrates the key transformations applied when jumping from `bar()` to `foo()` using `longjmp()` (with our original `keygen` pseudocode). The call to `setjmp()` is fine, since it returns normally. However, the call to `longjmp()` transfers to the `setjmp()` and acts as if `setjmp()` was returning again. So, the key is transformed from `bar`'s key to `longjmp`'s key, and then from `setjmp`'s key to `foo`'s key.

The transformation from `setjmp()` to `foo()` will fail since the key is still set to `longjmp`'s key. However, this problem is easily solved by giving `setjmp()` and `longjmp()` the same key. We can easily pre-install keys into the hashtable during Strata's initialization. If `setjmp()` and `longjmp()` have the same keys, a `longjmp()` will correctly maintain the global key.

4.4 Library Modifications

The key transformation system detailed here is useless without modifications to the library functions in order to verify that the correct key has been established. The source code to glibc was obtained so that the code could be modified to include key verification.

Checking the key is accomplished by a call to `strata_key_check()`, which takes the address of the current function as an argument. For example, protecting the `system()` function involves placing `strata_key_check(&system)` in the code for `system()`. The `strata_key_check()` function will look up the key for the address given in the argument and compare it to the global key. If the keys do not match, this is evidence that the intended control flow of the program has been disturbed. An error message is displayed and execution is terminated, preventing the attack.

5. Results

5.1 Verification

For verification, the Apache web server (version 1.3.33) was compiled with the modified compiler. There are no known buffer overflows in this version, so the source code was modified to add one. Work done by Shacham et. al. [5] describes a method for creating a buffer overflow in Apache and exploiting it with a return-to-libc attack. This same technique was replicated to produce a working exploit for testing and verification purposes.

The buffer overflow was created in Apache's `ap_getline()` function which returns the current line of an incoming HTTP request. A local buffer was added to this function, and the HTTP request is copied to it using `strcpy()`. A very large request will now overflow the local buffer and overwrite the stack frame. If the request is carefully constructed, the return address can be overwritten by `system`'s address with it's parameter pointing into the request string. As a result, any arbitrary string can be passed to `system()`.

With a little tweaking, the exploit was successful in executing `system()` with an unprotected version of Apache. Next the exploit was tried with the protected version of Apache using a version of `system()` that checks the key. This time, the exploit was not successful in executing system. The server logs showed that the key check had indeed failed while attempting to execute `system()`.

5.1 Performance

Performance was tested using the SPEC benchmark suite. Each benchmark was run using three different compilers. The first compiler, labeled *native*, is the Zephyr compiler using the unmodified lcc frontend. This version measures the performance of native execution on the processor. The second compiler, labeled *strata*, adds the default version of Strata. This version is running under Strata, but no protection is enabled. The final compiler, labeled *protected*, uses the full call sequence diversity protection. Currently, the only function with a key check is the `system()` function.

All benchmarks were compiled and executed on a dual 2.8GHz Intel Pentium 4 machine running Redhat Linux 7.3.

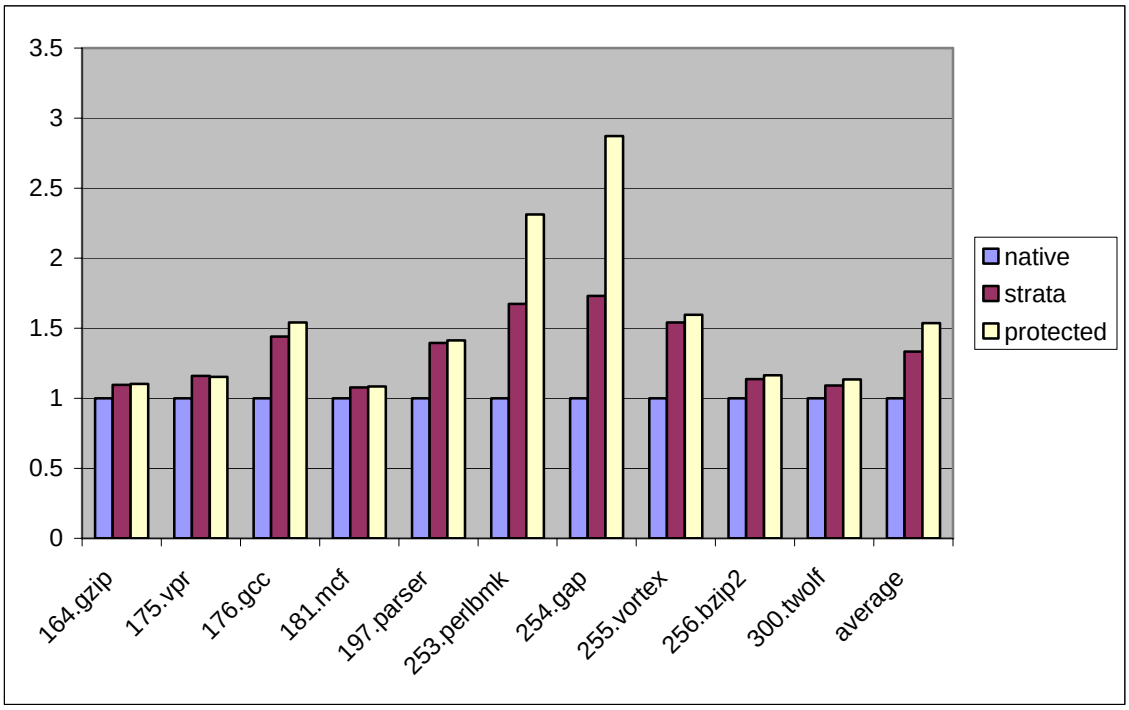


Figure 13: Overhead for SPEC benchmark suite normalized to native execution

Figure 13 shows the results of the performance tests. For the most part, the added protection of call sequence diversity does not add much more overhead than Strata itself. The gap and vortex benchmarks are the exceptions. These benchmarks add a much higher overhead presumably due to frequent indirect function calls. Even when you include these, there is an average overhead of 1.54x over native execution. Considering the overhead of software dynamic translation, this is reasonable. However, there is certainly room for improvement.

6. Related Work

Some techniques that defend against return-to-libc attacks involve what is known as address obfuscation. This includes the work of Bhatkar, DuVarney, and Sekar [6] and Xu, Kalbarczyk, and Iyer [7]. In this approach the locations of program data and code are randomized, as well as relative distances between data locations. The attacker will not be able to execute the attack because the locations of the stack and library code are randomized at load time. While the attacker could try to find these locations, the probability of guessing correctly is extremely low.

Several techniques have been proposed that protect the stack from being modified illegally. These include *StackGuard* [8] and *StackShield* [9]. *StackGuard* uses canary values on the stack that are checked when the program returns. *StackShield* saves the return address to a write-protected memory area when the function is entered. Such defenses would protect the return address from ever being modified. However, these defenses are not perfect and they do not address other methods of modifying control flow such as alterations to the global offset table.

7. Conclusion

When used in conjunction with techniques that prevent code injection, call sequence diversity can provide even further security by thwarting return-to-libc attacks. Any attempt to subvert the intended control flow of the program will be reflected in the global key. The program will not allow the execution of any protected function if the key is incorrect.

However, it is difficult to say exactly which functions should be protected. Certainly there are some obvious ones, such as `system()`, but it is difficult to say which functions can and cannot be used maliciously. It may be best to simply protect every function, though that approach may incur significantly higher overheads.

Given that a system call is generally necessary for a hacker to do any harm to the system, it may be a good idea to check the key at every system call. This could be accomplished by patching the operating system, or through Strata. An added bonus of this approach is that it could prevent most code injection attacks as well, since they generally rely on some sort of system call.

8. References

[1] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine 49*. 1996. http://www.phrack.org/phrack/49/P49-14

[2] Nergal. The advanced return-into-libc exploits: PaX case study. *Phrack Magazine 58*, Article 0x04. 2001. http://www.phrack.org/phrack/58/p58-0x04

[3] A. Appel, J. Davidson, and Ramsey N. The zephyr compiler infrastructure. November 1998. http://www.cs.virginia.edu/zephyr

[4] Kevin Scott and Jack Davidson. Strata: A Software Dynamic Translation Infrastructure, In Proceedings of the IEEE 2001 Workshop on Binary Translation, Barcelona, Spain, September 8, 2001.

[5] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. "On the Effectiveness of Address-Space Randomization." In B. Pfitzmann and Peng Liu, eds., CCS 2004, pp. 298-307. ACM Press, Oct. 2004.

[6] 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 12th USENIX Security Symposium*, pages 105-120, August 2003.

[7] Jun Xu, Zbigniew Kalbarczyk and Ravishankar K. Iyer. Transparent Runtime Randomization for Security. Technical Report UILU-ENG-03-2207, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign. May 2003.

[8] 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 7th USENIX Security Symposium*, Jan. 1998.

[9] Vendicator. Stack shield. *http://www.angelfire.com/sk/stackshield/*.

Appendix C: Genesis Fault Tree Analysis

Strata Fault Tree Analysis

Genesis Team

1.0 Introduction

Red team exercises can greatly increase knowledge about the vulnerabilities of a system. Generally speaking, red team exercises rapidly introduce adversarial thinking into the system testing process. This thinking centers on probing a system for vulnerabilities—system responses of advantage to an adversary—that may side-step or abuse the intended functionality of the system. The results of the red team exercise can lead to new insights into the system's service definition, improve its implementation, and provide more realistic expectations of its capabilities.

Despite the advantages of "red teaming," there are several constraints that often limit the effectiveness of the experience. First, experimental systems are often complex and contain novel, non-standard features. Second, red teams have limited time to examine a system, as their expertise is often chartered as a matter of cost effectiveness. This contrasts with the great length of time and effort that a true adversary might devote.

Therefore, we seek ways to enhance a red team's capabilities within a realistic time frame. We note the relationship between the novelty of many experimental systems and the limited time of procured red teams. Novelty means that time might be required for a red team to learn the system to be attacked. Yet time is of limited supply. We argue that by reducing the amount of time required to communicate expert knowledge of a system to a red team, the system team may enhance the red team exercise.

Communication of system knowledge to red teams is often informal. We propose the formalization of this communication through formal documentation to directly communicate system knowledge into the viewpoint of a red team. One such formal document is a fault tree analysis. This document describes the hazards, or potential undesirable scenarios that might arise in a system. Each hazard is followed by a logical postulate defining how the hazard can occur.

It is our intention that the presented Strata fault tree identify general strategies with which hazardous conditions may arise. The goal of this fault tree is not necessarily to list specific vulnerabilities, but to provide a guide for how one might track down and find vulnerabilities in a program protected by Strata's mechanisms.

1.1 Symbols

The fault tree uses several symbols for the purposes of brevity. These symbols have the following meaning:

S: Strata code, data structures, and control context

P: The server program's code, data structures, and control context

SP: The total Strata and server program's code, data structures, and control contexts (the entire application)

G: The specific goal of an attacker If an attacker achieves G, an attack succeeds. If not, it fails. Think of G as a Boolean postulate over the state of **SP**.

1.2 Textual Representation

The fault tree presented in the next section is represented as a tree as follows:

- The root of every tree is a hazard
- A node is labelled as one of LEAF, AND,OR, SUBTREE DEFINITION, or SUBTREE REFERENCE
- A LEAF NODE is a textual description of a fault
- An AND node indicates a fault such that the fault occurs only if all of its child nodes occur. Its textual representation is a summary of its causality or consequence
- An OR node indicates a fault that occurs if any of its child nodes occur. Its textual representation is summary or causally descriptive
- A SUBTREE definition node has a single child, and is simply a named subtree
- A Subtree reference node has no children, and indicates that the current node is a reference to a subtree as defined by a subtree node (see previous bullet.) A copy of the referenced subtree effectively replaces this node.
- A node's children are represented nodes below the node's representation in the list, and indented one unit.

2.0 Fault Tree Analysis

1. *Integrity Violation of code stream allows 'injected code' from an attack to achieve goal **G** in the strata system and its application*

•**AND:** *Integrity violation of code stream accomplishes **G** for attacker in **SP***
- •**OR:** *Integrity of code stream is compromised*
  - •**OR:** *Injected code is executed*
    - •**LEAF:** *Injected code is executed as system level code. OS/Hardware fault in protected system space integrity and/or authorization*
    - •**OR:** *Injected Code is executed as application level code*
      - •**AND:** *Code is injected into **SP** space and executed by SP code stream*
        - •**OR:** *Code is injected into **SP** space*
          - •**AND:** *Code is injected into **P**'s spaces*
            - •**OR:** *Input-based vulnerability exists in application **P***
              - •**LEAF:** *Stack overrun*
              - •**LEAF:** *Double-Free Malloc*
              - •**LEAF:** *Printf vulnerability*
              - •**LEAF:** *Other common input attack pathways not listed*
            - •**LEAF:** *Strata does not insert correct run-time check(s) for vulnerability(ies) into vulnerable fragment(s)*
          - •**OR:** *Code is injected into **S**'s spaces*

- **OR:** *Code is injected into Strata's fragment cache*
  - **OR:** *Insider injects malicious code into fragment cache*
    - **LEAF:** *Rogue **S** or **P** thread*
    - **LEAF:** *Privileged OS process modifies cache contents*
  - **AND:** *Fault in Strata translation/injection mechanism*
    - **LEAF:** *Design fault in Strata fragment cache injection responds to malicious payload program **P** to inject 'additional' unintended code. (ex. Short string literals stored by **P** in instruction cache contain attack code sequences, or help reverse engineer the encryption key)*
    - **LEAF:** *program **P** contains malicious payload*
  - **OR:** *Outside process modifies contents of fragment cache*
    - **OR:** *OS Permission Fault: Incorrect protection bits on instruction cache.*
    - **OR:** *Strata executer sets up Strata fragment cache for process sharing*
      - **LEAF:** *Erroneous process exec*
      - **LEAF:** *Malicious inside operator*
  - **LEAF:** *Code is stored in allocated IBTC or thread cache*
    - **LEAF:** *Feature is not in frequent write use in the program (example, no indirect jumps)*
    - **LEAF:** *Code is used before it is overwritten*

- **DEFINES SUBTREE A:**
  - **AND:** ***SP** Control Stream correctly executes injected code*
    - **OR:** *Code does not generate protection or instruction faults on execution*
    - **LEAF:** *Attacker guesses variations correctly on first try*

- **DEFINES SUBTREE E:** *SUBTREE*
  - **REFERENCE to SUBTREE:**
  - **AND:** *Memory Holding Injected Code is Executable*
    - **LEAF:** *MProtect cannot make read-allowed memory non-executable (This is always true in current STRATA implementation.)*
    - **LEAF:** *Code is allocated from POSIX mmap as executable by Strata in fragment cache*
    - **LEAF:** *MMAP fails to protect read|write only memory from execution (POSIX noncompliance)*
  - **OR:** ***SP** Control Stream is redirected to Code Injection Space*
    - **OR:** *By expected control flow of **SP** Injected code is reached by normal **SP** instruction/data stream.*

- **DEFINES SUBTREE B:**

- **•REFERENCE to SUBTREE:**. *(By modified control flow, Injected code is reached by abnormal **SP** instruction stream)*
- **•AND:** *Code is injected into ~**SP** space and executed by **SP** code stream*
  - **•LEAF:** *OS-Hardware Level Erroneous inter-process isolation*
  - **•REFERENCE to :** *SUBTREE A*
- **•DEFINES SUBTREE B:**
  - **•LEAF:** *Instruction pointer position is erroneous*
    - **•LEAF:** *Strata jumps to incorrect code position for next segment*
    - **•OR:** *Relative address is modified*
      - **•LEAF:** *return address modified*
      - **•LEAF:** *jump address modified*
      - **•LEAF:** *interrupt number modified*
    - **•LEAF:** *Absolute jump address is modified*
    - **•LEAF:** *Unintended interrupt is generated*
- **•LEAF:** *Unintended elements of instruction stream accomplish **G***

*2.Data manipulation affects behavior of **SP** allowing an attacker to achieve goal **G***

- **•AND:** *Data Overwrite modified behavior of **SP** to achieve goal **G***
- **•DEFINES SUBTREE E:**
  - **•REFERENCE to SUBTREE:**
  - **•DEFINES SUBTREE C:** *Data Overwrite write on **SP** space by adversary*
    - **•OR:** *Attacker changes (non-instructional) data of **SP***
      - **•LEAF:** *Attacker has legitimate indirect write access to a data structure through input messages (example, hash table degeneration)*
      - **•OR:** *Attacker applies a vulnerability in **SP** to write Strata /App data*
        - **•LEAF:** *Attacker's input exploit an error in a data structure implementation to modify contents within a data structure (adding extra records, duplicates, etc.)*
        - **•LEAF:** *Attacker's input exploits a general software architecture vulnerability allowing attack to succeed*
          - **•LEAF:** *Race conditions overwrite the legitimate client's request*
          - **•LEAF:** *Buffer overflows allow server-unintended modification of client request data*
          - **•LEAF:** *pointer-based attacks (double-free)*
          - **•LEAF:** *Code injected through a vulnerability deforms client request*
          - **•LEAF:** *Other Attack Strategies?*
      - **•AND:** *Incorrect setting of memory protection in OS, plus incorrect MPROTECT settings, allows another process to overwrite data*
        - **•LEAF:** *MProtect Fails to Set Read Only*
        - **•LEAF:** *Operating System allows another process access to the data by failure to use read protect (in mmap if POSIX OS)*
        - **•OR:** ***SP** data structure overwritten accidentally by another process, where that other process does so:*

- **LEAF:** *Accidentally*
- **LEAF:** *Maliciously as insider process*
- **LEAF:** *Unintentionally by vulnerability exploited by attacker(buffer overflow)*
- **LEAF:** *Carelessly through overextended power applied by attacker (printf)*

- **OR:** *Strata run-time checks do not detect the data overwrite*
  - **LEAF:** *No run-time check for data integrity*
  - **LEAF:** *Fault in run-time check*
  - **LEAF:** *false-negative in run-time check*
- **LEAF:** *The Overwritten Data results in the attacker achieving goal **G***
  - **LEAF:** *Overwritten data modifies branching behavior in code*
  - **LEAF:** *Overwritten data modifies internal records such as traces, alarms, profiles, etc.*
  - **LEAF:** *Overwritten data modifies values, sums, records of the application*
  - **LEAF:** *Overwritten data modifies who is contacted by the application (as might be used in an indirect DoS attack)*
  - **LEAF:** *others?*

3.The integrity of a client request is violated within the server environment and is not prevented from affecting the server. (Note the request includes any validation material. Signature, hash check, etc.)

- **DEFINES SUBTREE D:**
  - **AND:**
    - **OR:** *No Run-Time Detection*
      - **LEAF:** *Failure of all run-time checks on the integrity of client input.*
      - **LEAF:** *Run-time check cannot detect the change to input (ex, hash is modified to correctly reflect the input message)*
    - **LEAF:** *Integrity failure of client request is effective (not write over of same data)*
    - **OR:** *Memory storing the client request has modified values from client request*
      - **LEAF:** *Privileged insider modifies server memory*
      - **OR:** *MemProtect set inadequately on server, allowing the client request to be modified in the sever's memory by other processes with unintended access*
        - **LEAF:** *Accidentally*
        - **LEAF:** *Maliciously as insider process*
        - **LEAF:** *Unintentionally by vulnerability exploited by attacker(buffer overflow)*
        - **LEAF:** *Carelessly through overextended power applied by attacker (printf)*
      - **OR:** *Application and/or Strata code erroneously overwrites a client's message in local memory with erroneous data*

- **LEAF:** *Random, non malicious, Application/Strata error overwrites client request with predictable or unpredictable values*
- **OR:** *Deliberate outside influence results in placement of predictable (to outsider) values within the client request message*
  - **LEAF:** *Malicious outsider can send input to the system*
  - **OR:** *Malicious outsider can use its input events to cause **SP** code to unintentionally write over the other client's request through a vulnerability*
    - **LEAF:** *Race conditions overwrite the legitimate client's request*
    - **LEAF:** *Buffer overflows allow server-unintended modification of client request data*
    - **LEAF:** *pointer-based attacks (double-free)*
    - **LEAF:** *Code injected through a vulnerability deforms client request*
    - **LEAF:** *Other Attack Strategies not listed*
- **LEAF:** ***SP** code cannot decode the client message properly from local memory (The message is proper but **SP** has an interpretation error)*

4.The integrity of a bind response message to a client is violated within the server environment prior to sending to client

- **REFERENCE to SUBTREE:** ***C***

*5.Applying Strata to application causes erroneous application behavior achieving the goal **G** of an adversary (such as local denial of service to legitimate clients)*

- **OR:** *STRATA induces slow processing of application*
  - **OR:** *HASH TABLE DEGENERATION*
    - **AND:** *LARGE FRAGMENT CACHE DEGENERATION: causing linear search of the fragment cash for fragments each time a fragment is accessed.*
      - **LEAF:** *Odd fragment locations in executable memory does not cause a highly unusual executable in size that is noticed by sys admins*
      - **OR:** *BIND code is aligned in a manner that is degenerate with respect to the STRATA HASH() function*
        - **LEAF:** *Accidental misalignment (highly unlikely)*
        - **LEAF:** *Deliberate poor alignment (adversary has access to compiler)*
    - **LEAF:** *Indirect Branch Table Degeneration*
      - **LEAF:** *IBTC Resonance results in performance hit*
        - **LEAF:** *Adaptive Indirect Branch Caching is enabled*
        - **LEAF:** *Many Branch usages are highly oscillatory at a frequency resonating with the occurrence of reaching the branch usage count for storage in the IBTC (IBTC_THRESHOLD)*
        - **LEAF:** *The periodically used branches conflict with one another for space in the IBTC*

6.Privacy of client communications is violated

- **AND:** *Adversary reads information from **SP** memory and retrieves this information*
  - **LEAF:** *Adversary knows where messages to and from clients are stored*
  - **LEAF:** *Adversary reads the memory storing the client messages (read vulnerability)*
  - **LEAF:** *Adversary writes the information read into a useful location such as outgoing message queue (write vulnerability)*
  - **LEAF:** *Adversary intercepts the outgoing message*
  - **LEAF:** *Outgoing message is not encrypted (not secured) or encryption is known*

7.Strata reveals secrets necessary for many forms of attack

- **DEFINES SUBTREE E:**
  - **OR:** *Strata Configuration Information is revealed that is in use to protect the Application and Strata*
    - **AND:** *Encryption is known to the adversary*
      - **LEAF:** *Encryption is being used*
      - **OR:** *Encryption key is discovered*
        - **AND:** *Encryption key is read from Strata space by adversary*
          - **LEAF:** *Key is not read protected using M-protect when not in Strata-mode*
          - **LEAF:** *Key location is identified by the adversary*
        - **AND:** *Encryption key space is successfully searched by adversary*
          - **LEAF:** *Small enough encryption key space*
          - **LEAF:** *Application restart on crash*
          - **LEAF:** *repeated crashing is not easily detected*
      - **LEAF:** *Calling Sequence Permutation discovered by adversary*
      - **OR:** *Jump tag discovered by adversary*
        - **AND:** *Expected Jump Tag Read from Strata Space*
          - **LEAF:** *M-Protect does not protect Jump Tag in Strata Space when not in Strata execution*
          - **LEAF:** *Jump tag storage location is read by adversary and is known as jump tag to the adversary*
        - **OR:** *Jump Tag Read from Environment Space by adversary*
          - **LEAF:** *Through exploit allowing read of environment variables while executing application fragment*

8.Misconfiguration of Strata

- **OR:** *Misconfiguration of Strata*
  - **LEAF:** *Misconfiguration through operator error*
  - **LEAF:** *Misconfiguration for possibly by an attacker through re-execution of Strata through an libc exec vulnerability*

3.0 Potentially Vulnerable Data Structures

The data structures of greatest importance to Strata's security are:

- stats in strata.c
- target interface record
- Strata's thread table (it keeps its own)
- fcache: fragment cache that holds the decrypted application
- indirect branch targets cache
- Strata arena: memory management abstraction for Strata data structures
- Encryption Context Data Structure (stores the encryption key)

An adversary may examine how the integrity of these object's code and data may be violated so as to achieve the goal of the attacker, using the general guidelines of the presented fault tree.

4.0 Usage Suggestions

As was stated in the introduction to this document, the purpose of this fault tree is to identify general strategies with which hazardous conditions might arise. To the best of our ability the tree is sound, but cannot be complete. The goal of this fault tree is not to list specific vulnerabilities.To reiterate, *this is a guide for how one might track down and find vulnerabilities for Strata's protection of a server program, not a specification of a vulnerability's existence.*

Two approaches to using the tree seem useful.

1. *The high level hazards and their branches might provide guidelines, while the leaves of the tree give examples of how a search for a fault might proceed.*
2. *The leaves of the tree represent interactions with the data structures of interest to an attacker, and their correlation to hazards can be followed up through the tree*

Appendix D: Tamper Proofing

Novel Application of Genesis Technology.
Tamper Proofing White Paper

Tamperproofing Computer Software Using Strong Encryption

Department of Computer Science
University of Virginia
dependability@cs.virginia.edu
+1 434.982.2216

Background

Software tampering can be defined as carrying out unauthorized modifications on software that allow an adversary to misuse the software in some way. A survey of tampering issues and anti-tamper technology can be found in the work of Atallah et al [1].

Tampering is conducted by adversaries for many reasons including:

- Changing the software's functionality. For correct operation, all computer systems depend on the use of the software that was designed and built to realize the computer systems' intended purpose. If that software is altered or replaced by an adversary with malicious intent, the result could be serious. For example, information could be compromised or service could be altered. In a weapon system, an ATM machine, financial software, a "smart" card and similar systems, tremendous damage could be done.
- Reverse engineering the software. Software often contains valuable intellectual property that would be useful to an adversary. By stealing a copy of the software and reverse engineering it, the adversary can obtain the intellectual property with little cost.
- Changing the software's target. In some cases, reverse engineering is not necessary for an adversary to gain value from an existing piece of software, it is often only necessary to execute the software under conditions different from those intended by the software's owners. By stealing a copy and using his or her own target computer, an adversary gains the value of the software without paying for it. This type of malicious behaviour is often called piracy.

Since tampering can have serious consequences, the owners and operators of many computer systems desire a mechanism to make tampering as difficult as possible, i.e., they desire their software to be hardened against tampering, and, if possible, made tamperproof. The need for anti-tamper technology and practices has been documented by the Department of Defense [2] and the Government Accounting Office [3].

Tamperproofing software is difficult because the software is often stored at many different locations and often transmitted between locations. A given software system S might be built using hundreds of source-code files that are kept in a file system maintained by S's manufacturer. That file system will usually be shared so that a number of people might have

access to the file system and possibly also to all or part of S.

Once the system S is built, it will be in one of several different forms usually referred to as binary and be stored using one of several different media. Supplying the binary form of S to those who will use it might involve physical movement of the media or transmission over a network.

The binary software used by a computer is usually stored in a file system that is physically close to that computer. When it is not being used, the software remains available in that file system. When it is being used, the software is also stored in the main memory of the computer using it.

An adversary only needs to gain access to the software once in order to tamper with it, and, for some forms of tampering, the access gained need not be to all of the software. If the adversary wants to change the functionality of the software, all that he or she needs to do is gain access to that part of the software which provides the functionality to be changed. Access might be to the source files, the binary files, to the tools that are used to build the software (such as compilers and linkers), to shared libraries that the software uses, or to the software during execution. If the change is not detected, then the adversary has met his or her goal. The number of locations in which the software resides in its various forms makes protecting software from tampering very difficult.

The goal of those with a stake in the correct operation of the software is to ensure that the software is protected from tampering in all locations and in all forms. Protection of the software at the manufacturers location requires trust in all of those preparing the software. This is similar to any situation in which information is being developed, and so traditional techniques, such as access restriction, can be employed. Beyond the site of the software's original manufacturer, however, the problem of protecting the software against tampering is much harder since most people with access to the software are not known to be trustworthy.

Our technique described here achieves the stakeholders' goal and defeats all known credible tampering threats. It works by encrypting the software using a strong encryption algorithm. The protection that this affords is assured, and it is much more reliable as an anti-tampering technique than software obfuscation approaches. The technique implements anti-tampering efficiently requiring only a small execution overhead, can be applied to virtually any software system, and can be applied retroactively to existing systems.

Summary Of Technique

The technique meets the anti-tampering goal discussed above by maintaining the software in encrypted form until it is executed. The protection provided by encryption can be very strong because: (1) decryption by an adversary using state-space exploration requires resources that are beyond those available; and (2) decryption by an adversary using the appropriate key or keys is only possible if the key or keys are not protected properly. Existing techniques are available for key distribution and protection.

Encrypting software as an anti-tamper mechanism is not new. Present software encryption

mechanisms, however, either leave the software in plain form to such an extent that the software becomes vulnerable to tampering or the decryption process is extremely inefficient. The technique presented here addresses both of these problems.

Applying the technique consists of five steps: (1) the software is encrypted on a host computer in a trusted facility by its owners or the manufacturer prior to its deployment; (2) the software is conveyed to any location where it is needed in encrypted form; (3) the software is stored on the target computer upon which it is to run in encrypted form; (4) the software is loaded into memory on the target computer in encrypted form; (5) the software is decrypted just prior to execution. Only part of the software is kept in decrypted form at any given time. The decrypted software is held in a protected memory area.

Encryption at the trusted facility is carried out using an unspecified encryption mechanism. Decryption just prior to execution is effected using an unspecified decryption mechanism. An example of how decryption might be implemented in practice is the use of a supplemental specialized hardware unit of which many are available. Such devices contain the decryption key(s) and the processing hardware that executes the decryption algorithms. Without this device, the encrypted software cannot be decrypted. The keys used for encryption and decryption are made available to the host and target computers using a conventional key management system.

An example of how decryption might be controlled is by the use of a dynamic binary translation mechanism. With this approach, each fragment of the software is fetched as needed and sent to the decryption mechanism. The decrypted version of the fragment is stored in a region of memory called a fragment cache and then executed. If the fragment is executed more than once, the originally decrypted version is fetched from the fragment cache provided it is still there. The fragment cache is emptied periodically to ensure that only a small amount of the software is stored in plaintext form.

In order to tamper with the software after it has been encrypted, an adversary would have to either: (1) break the encryption; or (2) tamper with the software during execution. Decrypting the software is as difficult as decrypting any form of encrypted information. Provided the software is free of tampering when it is encrypted, the chances of tampering prior to execution is the same as the chances that the encryption can be broken.

Tampering during execution requires that the adversary gain access to that part of the software maintained in plain text form by the decryption mechanism. Nothing is specified in this technique about the decryption mechanism and so nothing is specified about what parts of the software will be in plain text form at any given point during execution. Using the example of a decryption mechanism given above in which dynamic binary translation is used, the only place where the software is maintained in plain text form is the fragment cache. In this example, the fragment cache is protected with a variety of software and hardware mechanisms.

Applying our Technique

Our technique provides very strong protection against tampering, for example:

- Changing the software’s functionality. This form of tampering is prevented by the fact that the

software remains encrypted everywhere that it is stored and during all transmissions prior to execution. Without the decryption key(s), any modification(s) effected by an adversary to the encrypted software would either not survive the decryption process or would be detected.

- Reverse engineering the software. This form of tampering is prevented by the fact that the software remains encrypted everywhere that it is stored and during all transmissions prior to execution. As a result, the adversary would only be able to acquire an encrypted version of the software. Acquiring the encrypted software does the adversary no good because he or she will not be able to conduct any form of static or dynamic analysis on the software.
- Changing the software's target. This form of tampering is prevented by the fact that the software requires a decryption key in order for it to be executed. Thus, copying the software will not allow it to be executed on an unauthorized target.

For more information, please refer to the following publication. While this paper focuses on using our technique for protection against code-injection attacks, the mechanisms involved apply to the general problem of anti-tampering.

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, and Jonathan Rowanhill. Second International Conference on Virtual Execution Environments. Ottawa, Canada, June 14-16, 2006.

The technology described in this paper provides a solid foundation on which to build sophisticated tamper proofing technologies, above and beyond what was described in this white paper.

References

[1] Atallah, M., E. Bryant, and M. Stytz, "A Survey of Anti-Tamper Technologies", CrossTalk: the Journal of Defense Software Engineering, November 2004.

[2] Office of the Secretary of Defense, Interim Defense Acquisition Guidebook, October 2002.

[3] Government Accounting Office, DOD Needs to Better Support Program Managers' Implementation of Anti-Tamper Protection, GAO-04-302, March 2004.

Appendix E: Secretless Security through Diversity

N-Variant Systems: A Secretless Framework for Security through Diversity

N-Variant Systems
A Secretless Framework for Security through Diversity

Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill,
Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser
University of Virginia, Department of Computer Science
http://www.nvariant.org

Abstract

We present an architectural framework for systematically using automated diversity to provide high assurance detection and disruption for large classes of attacks. The framework executes a set of automatically diversified variants on the same inputs, and monitors their behavior to detect divergences. The benefit of this approach is that it requires an attacker to simultaneously compromise all system variants with the same input. By constructing variants with disjoint exploitation sets, we can make it impossible to carry out large classes of important attacks. In contrast to previous approaches that use automated diversity for security, our approach does not rely on keeping any secrets. In this paper, we introduce the N-variant systems framework, present a model for analyzing security properties of N-variant systems, define variations that can be used to detect attacks that involve referencing absolute memory addresses and executing injected code, and describe and present performance results from a prototype implementation.

1. Introduction

Many security researchers have noted that the current computing monoculture leaves our infrastructure vulnerable to a massive, rapid attack [70, 29, 59]. One mitigation strategy that has been proposed is to increase software diversity. By making systems appear different to attackers, diversity makes it more difficult to construct exploits and limits an attack's ability to propagate. Several techniques for automatically producing diversity have been developed including rearranging memory [8, 26, 25, 69] and randomizing the instruction set [6, 35]. All these techniques depend on keeping certain properties of the running execution secret from the attacker. Typically, these properties are determined by a secret key used to control the randomization. If the secret used to produce a given variant is compromised, an attack can be constructed that successfully attacks that variant. Pointer obfuscation techniques, memory address space randomization, and instruction set randomization have all been demonstrated to be vulnerable to remote attacks [55, 58, 64]. Further, the diversification secret may be compromised through side channels, insufficient entropy, or insider attacks.

Our work uses artificial diversity in a new way that does not depend on keeping secrets: instead of diversifying individual systems, we construct a single system containing multiple variants designed to have disjoint exploitation sets. Figure 1 illustrates our framework. We refer to the entire server as an N-variant system. The system shown is a 2-variant system, but our framework generalizes to any number of variants. The polygrapher takes input from the client and copies it to all the variants. The original server process P is replaced with the two variants, P_0 and P_1. The variants maintain the client-observable behavior of P on all normal inputs. They are, however, artificially diversified in a way that makes them behave differently on abnormal inputs that correspond to an attack of a certain class. The monitor observes the behavior of the variants to detect divergences which reveal attacks. When a divergence is detected, the monitor restarts the variants in known uncompromised states.

As a simple example, suppose P_0 and P_1 use disjoint memory spaces such that any absolute memory address that is valid in P_0 is invalid in P_1, and vice versa. Since

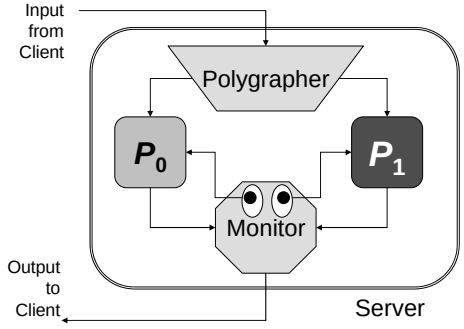


Figure 1. N-Variant System Framework.

the variants are transformed to provide the same semantics regardless of the memory space used, the behavior on all normal inputs is identical (assuming deterministic behavior, which we address in Section 5). However, if an exploit uses an absolute memory address directly, it must be an invalid address on one of the two variants. The monitor can easily detect the illegal memory access on the other variant since it is detected automatically by the operating system. When monitoring is done at the system call level, as in our prototype implementation, the attack is detected before any external state is modified or output is returned to the attacker.

The key insight behind our approach is that in order for an attacker to exploit a vulnerability in *P*, a pathway must exist on one of the variants that exploits the vulnerability without producing detectably anomalous behavior on any of the other variants. If no such pathway exists, there is no way for the attacker to construct a successful attack, even if the attacker has complete knowledge of the variants. Removing the need to keep secrets means we do not need to be concerned with probing or guessing attacks, or even with attacks that take advantage of insider information.

Our key contributions are:

1. Introducing the N-variant systems framework that uses automated diversity techniques to provide high assurance security properties without needing to keep any secrets.
2. Developing a model for reasoning about N-variant systems including the definition of the normal equivalence and detection properties used to prove security properties of an ideal N-variant system (Section 3).
3. Identifying two example techniques for providing variation in N-variant systems: the memory address partitioning technique (introduced above) that detects attacks that involve absolute memory references and the instruction tagging technique that detects attempts to execute injected code (Section 4).
4. Describing a Linux kernel system implementation and analyzing its performance (Section 5).

In this paper we do not address recovery but consider it to be a successful outcome when our system transforms an attack that could compromise privacy and integrity into an attack that at worst causes a service shutdown that denies service to legitimate users. It has not escaped our attention, however, that examining differences between the states of the two variants at the point when an attack is detected provides some intriguing recovery possibilities. Section 6 speculates on these opportunities and other possible extensions to our work.

2. Related Work

There has been extensive work done on eliminating security vulnerabilities and mitigating attacks. Here, we briefly describe previous work on other types of defenses and automated diversity, and summarize related work on redundant processing and design diversity frameworks.

Other defenses. Many of the specific vulnerabilities we address have well known elimination, mitigation and disruption techniques. Buffer overflows have been widely studied and numerous defenses have been developed including static analysis to detect and eliminate the vulnerabilities [66, 67, 39, 23], program transformation and dynamic detection techniques [19, 5, 30, 45, 49, 57] and hardware modifications [38, 40, 41, 64]. There have also been several defenses proposed for string format vulnerabilities [56, 20, 63, 47]. Some of these techniques can mitigate specific classes of vulnerabilities with less expense and performance overhead than is required for our approach. Specific defenses, however, only prevent a limited class of specific vulnerabilities. Our approach is more general; it can mitigate all attacks that depend on particular functionality such as injecting code or accessing absolute addresses.

More general defenses have been proposed for some attack classes. For example, no execute pages (as provided by OpenBSD's W^X and Windows XP Service Pack 2) prevent many code injection attacks [2], dynamic taint analysis tracks information flow to identify memory corruption attacks [43], and control-flow integrity can detect attacks that corrupt an application to follow invalid execution paths [1]. Although these are promising approaches, they are limited to particular attack classes. Our framework is more general in the sense that we can construct defense against any attacker capability that can be varied across variants in an N-variant system.

Automated diversity. Automated diversity applies transformations to software to increase the difficulty an attacker will face in exploiting a security vulnerability in that software. Numerous transformation techniques have been proposed including rearranging memory [26,

8, 69, 25], randomizing system calls [17], and randomizing the instruction set [6, 35]. Our work is complementary to work on producing diversity; we can incorporate many different sources of variation as long as variants are constructed carefully to ensure the disjointedness required by our framework. A major advantage of the N-variant systems approach is that we do not rely on secrets for our security properties. This means we can employ diversification techniques with low entropy, so long as the transformations are able to produce variants with disjoint exploitation sets. Holland, Lim, and Seltzer propose many low entropy diversification techniques including number representations, register sets, stack direction, and memory layout [31]. In addition, our approach is not vulnerable to the type of secret-breaking attacks that have been demonstrated against secret-based diversity defenses [55, 58, 64].

O'Donnell and Sethu studied techniques for distributing diversity at the level of different software packages in a network to mitigate spreading attacks [44]. This can limit the ability of a worm exploiting a vulnerability present in only one of the software packages to spread on a network. Unlike our approach, however, even at the network level an attacker who discovers vulnerabilities in more than one of the software packages can exploit each of them independently.

Redundant execution. The idea of using redundant program executions for various purposes is not a new one. Architectures involving replicated processes have been proposed as a means to aid debugging, to provide fault tolerance, to improve dependability, and more recently, to harden vulnerable services against attacks.

The earliest work to consider running multiple variants of a process of which we are aware is Knowlton's 1968 paper [37] on a variant technique for detecting and localizing programming errors. It proposed simultaneously executing two programs which were logically equivalent but assembled differently by breaking the code into fragments, and then reordering the code fragments and data segments with appropriate jump instructions inserted between code fragments to preserve the original program semantics. The CPU could run in a checking mode that would execute both programs in parallel and verify that they execute semantically equivalent instructions. The variants they used did not provide any guarantees, but provided a high probability of detecting many programming errors such as out-of-range control transfers and wild memory fetches.

More recently, Berger and Zorn proposed a redundant execution framework with multiple replicas each with a different randomized layout of objects within the heap to provide probabilistic memory safety [7]. Since there is no guarantee that there will not be references at the same absolute locations, or reachable through the same relative offsets, their approach can provide only probabilistic expectations that a memory corruption will be detected by producing noticeably different behavior on the variants. Their goals were to enhance reliability and availability, rather than to detect and resist attacks. Consequently, when variations diverge in their framework, they allow the agreeing replicas to continue based on the assumption that the cause of the divergence in the other replicas was due a memory flaw rather than a successful attack. Their replication framework only handles processes whose I/O is through standard in/out, and only a limited number of system calls are caught in user space to ensure all replicas see the same values. Since monitoring is only on the standard output, a compromised replica could be successfully performing an attack and, as long as it does not fill up its standard out buffer, the monitor would not notice. The key difference between their approach and ours, is that their approach is probabilistic whereas our variants are constructed to guarantee disjointedness with respect to some property, and thereby can provide guarantees of invulnerability to particular attack classes. A possible extension to our work would consider variations providing probabilistic protection, such as the heap randomization technique they used, to deal with attack classes for which disjointedness is infeasible.

Redundant processing of the same instruction stream by multiple processors has been used as a way to provide fault-tolerance by Stratus [68] and Tandem [32] computers. For example, Integrity S2 used triple redundancy in hardware with three synchronized identical processors executing the same instructions [32]. A majority voter selects the majority output from the three processors, and a vote analyzer compares the outputs to activate a failure mode when a divergence is detected. This type of redundancy provides resilience to hardware faults, but no protection against malicious attacks that exploit vulnerabilities in the software, which is identical on all three processors. Slipstream processors are an interesting variation of this, where two redundant versions of the instruction stream execute, but instructions that are dynamically determined to be likely to be unnecessary are removed from the first stream which executes speculatively [60]. The second stream executes behind the first stream, and the processor detects inconsistencies between the two

executions. These deviations either indicate false predications about unnecessary computations (such as a mispredicted branch) or hardware faults.

The distributed systems community has used active replication to achieve fault tolerance [9, 10, 16, 18, 50]. With active replication, all replicas are running the same software and process the same requests. Unlike our approach, however, active replication does nothing to hide design flaws in the software since all replicas are running the same software. To mitigate this problem, Schneider and Zhou have suggested proactive diversity, a technique for periodically randomizing replicas to justify the assumption that server replicas fail independently and to limit the window of vulnerability in which replicas are susceptible to the same exploit [51]. Active replication and N-variant systems are complementary approaches. Combining them can provide the benefits of both approaches with the overhead and costs associated with either approach independently.

Design diversity frameworks. The name *N-variant systems* is inspired by, but fundamentally different from, the technique known as *N-version programming* [3, 14]. The N-version programming method uses several independent development groups to develop different implementations of the same specification with the hope that different development groups will produce versions without common faults. The use of N-version programming to help with system security was proposed by Joseph [33]. He analyzed design diversity as manifest in N-version programming to see whether it could defeat certain attacks and developed an analogy between faults in computing systems that might affect reliability and vulnerabilities in computer systems that might affect security. He argued that N-version programming techniques might allow vulnerabilities to be masked. However, N-version programming provides no guarantee that the versions produced by different teams will not have common flaws. Indeed, experiments have shown that common flaws in implementations do occur [36]. In our work, program variants are created by mechanical transformations engineered specifically to differ in particular ways that enable attack detection. In addition, our variants are produced mechanically, so the cost of multiple development teams is avoided.

Three recent projects [46, 62, 28] have explored using design diversity in architectures similar to the one we propose here in which the outputs or behaviors of two diverse implementations of the same service (e.g., HTTP servers Apache on Linux and IIS on Windows) are compared and differences above a set threshold indicate a likely attack. The key difference between those projects and our work is that whereas they use diverse available implementations of the same service, we use techniques to artificially produce specific kinds of variation. The HACQIT project [34, 46] deployed two COTS web servers (IIS running on Windows and Apache running on Linux) in an architecture where a third computer forwarded all requests to both servers and compared their responses. A divergence was detected when the HTTP status code differed, hence divergences that caused the servers to modify external state differently or produce different output pages would not be detected. The system described by Totel, Majorczyk, and Mé extended this idea to compare the actual web page responses of the two servers [62]. Since different servers do not produce exactly the same output on all non-attack requests because of nondeterminism, design differences in the servers, and host-specific properties, they developed an algorithm that compares a set of server responses to determine which divergences are likely to correspond to attacks and which are benign. The system proposed by Gao, Reiter, and Song [28] deployed multiple servers in a similar way, but monitored their behavior using a distance metric that examined the sequence of system calls each server made to determine when the server behaviors diverged beyond a threshold amount.

All of these systems use multiple available implementations of the same service running on isolated machines and compare the output or aspects of the behavior to notice when the servers diverged. They differ in their system architectures and in how divergences are recognized. The primary advantage of our work over these approaches is the level of assurance automated diversity and monitoring can provide over design diversity. Because our system takes advantage of knowing exactly how the variants differ, we can make security claims about large attack classes. With design diversity, security claims depend on the implementations being sufficiently different to diverge noticeably on the attack (and functionality claims depend on the behaviors being sufficiently similar not exceed the divergence threshold on non-attack inputs). In addition, these approaches can be used only when diverse implementations of the same service are available. For HTTP servers, this is the case, but for custom servers the costs of producing a diverse implementation are prohibitive in most cases. Further, even though many HTTP servers exist, most advanced websites take advantages of server-specific functionality (such as server-side includes provided by Apache), so would not work on an alternate server.

Design diversity approaches offer the advantage that they may be able to detect attacks that are at the level of application semantics rather than low-level memory corruption or code injection attacks that are better detected by artificial diversity. In Section 6, we consider possible extensions to our work that would combine both approaches to provide defenses against both types of attacks.

3. Model

Our goal is to show that for all attacks in a particular attack class, if one variant is compromised by a given attack, another variant must exhibit divergent behavior that is detected by the monitor. To show this, we develop a model of execution for an N-variant system and define two properties the variant processes must maintain to provide a detection guarantee.

We can view an execution as a possibly infinite sequence of states: $[S_0, S_1, \ldots]$. In an N-variant system, the state of the system can be represented using a tuple of the states of the variants (for simplicity, this argument assumes the polygrapher and monitor are stateless; in our implementation, they do maintain some state but we ignore that in this presentation). Hence, an execution of an N-variant system is a sequence of state-tuples where $S_{t,v}$ represents the state of variant v at step t: $[<S_{0,0}, S_{0,1}, \ldots S_{0,N-1}>, <S_{1,0}, S_{1,1}, \ldots S_{1,N-1}>, \ldots]$.

Because of the artificial variation, the concrete state of each variant differs. Each variant has a *canonicalization function*, C_v, that maps its state to a canonical state that matches the corresponding state for the original process. For example, if the variation alters memory addresses, the mapping function would need to map the variant's altered addresses to canonical addresses. Under normal execution, at every execution step the canonicalized states of all variants are identical to the original program state:

$$\forall t \geq 0,\ 0 \leq v < N,\ 0 \leq w < N:$$
$$C_v(S_{t,v}) = C_w(S_{t,w}) = S_t.$$

Each variant has a *transition function*, T_v, that takes a state and an input and produces the next state. The original program, P, also has a transition function, T. The set of possible transitions can be partitioned into *consistent transitions* and *aberrant transitions*. Consistent transitions take the system from one normal state to another normal state; aberrant transitions take the system from a normal state to a compromised state. An attack is successful if it produces an aberrant transition without detection. Our goal is to detect all aberrant transitions.

We partition possible variant states into three sets: *normal*, *compromised*, and *alarm*. A variant in a normal state is behaving as intended. A variant in a compromised state has been successfully compromised by a malicious attack. A variant in an alarm state is anomalous in a way that is detectable by the monitor. We aim to guarantee that the N-variant system never enters a state-tuple that contains one or more variants in comprised states without any variants in alarm states. To establish this we need two properties: *normal equivalence* and *detection*.

Normal equivalence. The normal equivalence property is satisfied if the N-variant system synchronizes the states of all variants. That is, whenever all variants are in normal states, they must be in states that correspond to the same canonical state. For security, it is sufficient to show the variants remain in equivalent states. For correctness, we would also like to know the canonical state of each of the variants is equivalent to the state of the original process.

We can prove the normal equivalence property statically using induction:

1. Show that initially all variants are in the same canonical state: $\forall\ 0 \leq v < N$: $C_i(S_{0,v}) = S_0$.
2. Show that every normal transition preserves the equivalence when the system is initially in a normal state:

$$\forall S \in Normal,\ 0 \leq v < N,\ S_v$$
$$\text{where } C_v(S_v) = S,\ p \in Inputs:$$
$$C_v(T_v(S_v, p)) = T(S, p).$$

Alternatively, we can establish it dynamically by examining the states of the variants and using the canonicalization function to check the variants are in equivalent states after every step. In practice, neither a full static proof nor a complete dynamic comparison is likely to be feasible for real systems. Instead, we argue that our implementation provides a limited form of normal equivalence using a combination of static argument and limited dynamic comparison, as we discuss in Section 5.

Detection. The detection property guarantees that all attacks in a certain class will be detected by the N-variant system as long as the normal equivalence property is satisfied. To establish the detection property, we need to know that any input that causes one variant to enter a compromised state must also cause some other variant to enter an alarm state. Because of the normal equivalence property, we can assume the

variants all are in equivalent states before processing this input. Thus, we need to show:

$\forall S \in Normal, 0 \leq v < N, S_v$ where $C_v(S_v) = S$,
$\forall p \in Inputs$:
$T_v(S_v, p) \in Compromised \Rightarrow$
$\exists w$ such that $T_w(S_w, p) \in Alarm$ and $C_w(S_w) = S$

If the detection property is established, we know that whenever one of the variants enters a compromised state, one of the variants must enter an alarm state. An ideal monitor would instantly detect the alarm state and prevent all the other variants from continuing. This would guarantee that the system never operates in a state in which any variant is compromised.

In practice, building such a monitor is impossible since we cannot keep the variants perfectly synchronized or detect alarm states instantly. However, we can approximate this behavior by delaying any external effects (including responses to the client) until all variants have passed a critical point. This keeps the variants loosely synchronized, and approximates the behavior of instantly terminating all other variants when one variant encounters an alarm state. It leaves open the possibility that a compromised variant could corrupt the state of other parts of the system (including the monitor and other variants) before the alarm state is detected. An implementation must use isolation mechanisms to limit this possibility.

4. Variations

Our framework works with any diversification technique that produces variants different enough to provide detection of a class of attack but similar enough to establish a normal equivalence property. The variation used to diversify the variants determines the attack class the N-variant system can detect. The detection property is defined by the class of attack we detect, so we will consider attack classes, such as attacks that involve executing injected instructions, rather than vulnerability classes such as buffer overflow vulnerabilities.

Next, we describe two variations we have implemented: address space partitioning and instruction set tagging. We argue (informally) that they satisfy both the normal equivalence property and the detection condition for important classes of attacks. The framework is general enough to support many other possible variations, which we plan to explore in future work. Other possible variations that could provide useful security properties include varying memory organization, file naming, scheduling, system calls, calling conventions, configuration properties, and the root user id.

4.1 Address Space Partitioning

The Introduction described an example variation where the address space is partitioned between two variants to disrupt attacks that rely on absolute addresses. This simple variation does not prevent all memory corruption attacks since some attacks depend only on relative addressing, but it does prevent all memory corruption attacks that involve direct references to absolute addresses. Several common vulnerabilities including format string [56, 54], integer overflow, and double-free [24] may allow an attacker to overwrite an absolute location in the target's address space. This opportunity can be exploited to give an attacker control of a process, for example, by modifying the Global Offset Table [24] or the .dtors segment of an ELF executable [48]. Regardless of the vulnerability exploited and the targeted data structure, if the attack depends on loading or storing to an absolute address it will be detected by our partitioning variants. Since the variation alters absolute addresses, it is necessary that the original program does not depend on actual memory addresses (for example, using the value of a pointer directly in a decision). Although it is easy to construct programs that do not satisfy this property, most sensible programs should not depend on actual memory addresses.

Detection. Suppose P_0 only uses addresses whose high bit is 0 and P_1 only uses addresses whose high bit is 1. We can map the normal state of P_0 and P_1 to equivalent states using the identity function for C_0 and a function that flips the high bit of all memory addresses for C_1 (to map onto the actual addresses used by P, more complex mapping functions may be needed). The transition functions, T_0 and T_1 are identical; the generated code is what makes things different since a different address will be referenced in the generated code for any absolute address reference. If an attack involves referencing an absolute address, the attacker must choose an address whose high bit is either a 0 or 1. If it is a 0, then P_0 may transition to a compromised state, but P_1 will transition to an alarm state when it attempts to access a memory address outside P_1's address space. In Unix systems, this alarm state is detected by the operating system as a segmentation fault. Conversely, if the attacker chooses an address whose high bit is 1, P_1 may be compromised but P_0 must enter an alarm state. In either case, the monitor detects the compromise and prevents any external state modifications including output transmission to the client.

Our detection argument relies on the assumption that the attacker must construct the entire address directly. For most scenarios, this assumption is likely to be valid. For certain vulnerabilities on platforms that are not byte-aligned, however, it may not be. If the attacker is able to overwrite an existing address in the program without overwriting the high bit, the attacker may be able to construct an address that is valid in both variants. Similarly, if an attacker can corrupt a value that is subsequently used with a transformed absolute address in an address calculation, the detection property is violated. As with relative attacks, this indirect memory attacks would not be detected by this variation.

Normal equivalence. We have two options for establishing the normal equivalence property: we can check it dynamically using the monitor, or we can prove it statically by analyzing the variants. A pure dynamic approach is attractive for security assurance because of its simplicity but impractical for performance-critical servers. The monitor would need to implement C_0 and C_1 and compute the canonical states of each variant at the end of each instruction execution. If the states match, normal equivalence is satisfied. In practice, however, this approach is likely to be prohibitively expensive. We can optimize the check by limiting the comparison to the subset of the execution state that may have changed and only checking the state after particular instructions, but the overhead of checking the states of the variants after every step will still be unacceptable for most services.

The static approach requires proving that for every possible normal state, all normal transitions result in equivalent states on the two variants. This property requires that no instruction in P can distinguish between the two variants. For example, if there were a conditional jump in P that depended on the high bit of the address of some variable, P_0 and P_1 would end up in different states after executing that instruction. An attacker could take advantage of such an opportunity to get the variants in different states such that an input that transitions P_0 to a compromised state does not cause P_1 to reach an alarm state. For example, if the divergence is used to put P_0 in a state where the next client input will be passed to a vulnerable string format call, but the next client input to P_1 is processed harmlessly by some other code, an attacker may be able to successfully compromise the N-variant system. A divergence could also occur if some part of the system is nondeterministic, and the operating environment does not eliminate this nondeterminism (see Section 5). Finally, if P is vulnerable to some other class of attack, such as code injection, an attacker may be able to alter the transition functions T_0 and T_1 in a way that allows the memory corruption attack to be exploited differently on the two variants to avoid detection (of course, an attacker who can inject code can already compromise the system in arbitrary ways).

In practice, it will not usually be possible to completely establish normal equivalence statically for real systems but rather we will use a combination of static and dynamic arguments, along with assumptions about the target service. A combination of static and dynamic techniques for checking equivalence may be able to provide higher assurance without the overhead necessary for full dynamic equivalence checking. Our prototype implementation checks equivalence dynamically at the level of system calls, but relies on informal static arguments to establish equivalence between them.

Implementation. To partition the address space, we vary the location of the application data and code segments. The memory addresses used by P_0 and P_1 are disjoint: any data address that is valid for P_0 is invalid for P_1, and vice versa. We use a linker script to create the two variants. Each variant loads both the code and data segments of the variants at different starting addresses from the other variant. To ensure that their sets of valid data memory addresses are disjoint, we use ulimit to limit the size of P_0's data segment so it cannot grow to overlap P_1's address space.

4.2 Instruction Set Tagging

Whereas partitioning the memory address space disrupts a class of memory corruption attacks, partitioning the instruction set disrupts code injection attacks. There are several possible ways to partition the instruction set.

One possibility would be to execute the variants on different processors, for example one variant could run on an x86 and the other on a PowerPC. Establishing the security of such an approach would be very difficult, however. To obtain the normal equivalence property we would need a way of mapping the concrete states of the different machines to a common state. Worse, to obtain the detection property, we would need to prove that no string of bits that corresponds to a successful malicious attack on one instruction set and a valid instruction sequence on the other instruction set. Although it is likely that most sequences of malicious x86 instructions contain an invalid PowerPC instruction, it is certainly possible for attackers to design instruction sequences that are valid on both platforms (although we are not

aware of any programs that do this for the x86 and PowerPC, Sjoerd Mullender and Robbert van Renesse won the 1984 International Obfuscated C Code Contest with an entry that replaced main with an array of bytes that was valid machine code for both the Vax and PDP-11 but executed differently on each platform [35]).

Instead, we use a single instruction set but prepend a variant-specific tag to all instructions. The diversification transformation takes *P* and inserts the appropriate tag bit before each instruction to produce each variant.

Detection. The variation detects any attack that involves executing injected code, as long as the mechanism used to inject code involves injecting complete instructions. If memory is bit-addressable, an attacker could overwrite just the part of the instruction after the tag bit, thereby changing an existing instruction while preserving the original tag bit. If the attacker can inject the intended code in memory, and then have the program execute code already in the executable that transforms the injected memory (for example, by XORing each byte with a constant that is different in the two variants), then it is conceivable that an attacker could execute an indirect code injection attack where the code is transformed differently on the two variants before executing to evade the detection property. For all known realistic code injection attacks, neither of these is considered a serious risk.

Normal equivalence. The only difference between the two variants is the instruction tag, which has no effect on instruction execution. The variants could diverge, however, if the program examines its own instructions and makes decisions that depend on the tag. It is unlikely that a non-malicious program would do this. As with the memory partitioning, if the instruction tags are visible to the executing process an attacker might be able to make them execute code that depends on the instruction tags to cause the variants to diverge before launching the code injection attack on one of the variants. To prevent this, we need to store the tagged instructions in memory that is not readable to the executing process and remove the tags before those instructions reach the processor.

Implementation. To implement instruction set tagging, we use a combination of binary rewriting before execution and software dynamic translation during execution. We use Diablo [61, 22], a retargetable binary rewriting framework, to insert the tags. Diablo provides mechanisms for modifying an x86 binary in ELF format. We use these to insert the appropriate variant-specific tag before every instruction. For simplicity, we use a full byte tag even though a single bit would suffice for two variants. There is no need to keep the tags secret, just that they are different; we use 10101010 and 01010101 for the *A* and *B* variant tags.

At run-time, the tags are checked and removed before instructions reach the processor. This is done using Strata, a software dynamic translation tool [52, 53]. Strata and other software dynamic translators [4, 11] have demonstrated that it is possible to implement software dynamic translation without unreasonable performance penalty. In our experiments (Section 5), Strata's overhead is only a few percent. The Strata VM mediates application execution by examining and translating instructions before they execute on the host CPU. Translated instructions are placed in the fragment cache and then executed directly on the host CPU. Before switching to the application code, the Strata VM uses mprotect to protect critical data structures including the fragment cache from being overwritten by the application. At the end of a translated block, Strata appends trampoline code that will switch execution back to the Strata VM, passing in the next application PC so that the next fragment can be translated and execution will continue. We implement the instruction set tagging by extending Strata's instruction fetch module. The modified instruction fetch module checks that the fetched instruction has the correct tag for this variant; if it does not, a security violation is detected and execution terminates. Otherwise, it removes the instruction tag before placing the actual instruction in the fragment cache. The code executing on the host processor contains no tags and can execute normally.

5. Framework Implementation

Implementing an N-variant system involves generating variants such as those described in Section 4 as well as implementing the polygrapher and monitor. The trusted computing base comprises the polygrapher, monitor and mechanisms used to produce the variants, as well as any operating system functionality that is common across the variants. An overriding constraint on our design is that it be fully automated. Any technique that requires manual modification of the server to create variants or application-specific monitoring would impose too large a deployment burden to be used widely. To enable rapid development, our implementations are entirely in software. Hardware implementations would have security and performance advantages, especially in monitoring the instruction tags. Furthermore, placing monitoring as close as possible to the processor eliminates the risk that an

attacker can exploit a vulnerability in the monitoring mechanism to inject instructions between the enforcement mechanism and the processor.

The design space for N-variant systems implementations presents a challenging trade-off between isolation of the variants, polygrapher, and monitor and the need to keep the variant processes synchronized enough to establish the normal equivalence property. The other main design decision is the granularity of the monitoring. Ideally, the complete state of each variant would be inspected after each instruction. For performance reasons, however, we can only observe aspects of the state at key execution points. Incomplete monitoring means that an attacker may be able to exploit a different vulnerability in the server to violate the normal equivalence property, thereby enabling an attack that would have otherwise been detected to be carried out without detection. For example, an attacker could exploit a race condition in the server to make the variants diverge in ways that are not detected by the monitor. Once the variants have diverged, the attacker can construct an input that exploits the vulnerability in one variant, but does not produce the detected alarm state on the other variants because they started from different states.

In our first proof-of-concept implementation, described in Section 5.1, we emphasized isolation and executed the variants on separate machines. This meant that any nondeterminism in the server program or aspects of the host state visible to the server program that differed between the machines could be exploited by an attacker to cause the processes to diverge and then allow a successful attack. It also meant the monitor only observed the outputs produced by the two variants that would be sent over the network. This enabled certain attacks to be detected, but meant a motivated attacker could cause the states to diverge in ways that were not visible from the output (such as corrupting server data) but still achieved the attacker's goals.

Our experience with this implementation led us to conclude that a general N-variant systems framework needed closer integration of the variant processes to prevent arbitrary divergences. We developed such a framework as a kernel modification that allows multiple variants to run on the same platform and normal equivalence to be established at system call granularity. This eliminates most causes of nondeterminism and improves the performance of the overall system. Section 5.2 describes our Linux kernel implementation, and Section 5.3 presents performance results running Apache variants on our system.

5.1 Proof-of-Concept Implementation

In our proof-of-concept implementation, the variants are isolated on separate machines and the polygrapher and monitor are both implemented by the nvd process running on its own machine. We used our implementation to protect both a toy server we constructed and Apache. In order for our approach to work in practice it is essential that no manual modification to the server source code is necessary. Hence, each server variant must execute in a context where it appears to be interacting normally with the client. We accomplish this by using divert sockets to give each variant the illusion that it is interacting directly with a normal client. To implement the polygrapher we use ipfw, a firewall implementation for FreeBSD [27] with a rule that redirects packets on port 80 (HTTP server) to our nvd process which adjusts the TCP sequence numbers to be consistent with the variant's numbering. Instead of sending responses directly to the client, the variant's responses are diverted back to nvd, which buffers the responses from all of the variants. The responses from P_0 are transmitted back to the client only if a comparably long response is also received from the other variants. Hence, if any variant crashes on a client input, the response is never sent to the client and nvd restarts the server in a known uncompromised state.

We tested our system by using it to protect a toy server we constructed with a simple vulnerability and Apache, and attempted to compromise those servers using previously known exploits as well as constructed exploits designed to attack a particular variant. Exploit testing does not provide any guarantees of the security of our system, of course, but it does demonstrate that the correct behavior happens under the tested conditions to increase our confidence in our approach and implementation. Our toy server contained a contrived format string vulnerability, and we developed an exploit that used that vulnerability to write to an arbitrary memory address. The exploit could be customized to work against either variation, but against the N-variant system both versions would lead to one of the variants crashing. The monitor detects the crash and prevents compromised outputs from reaching the client. We also tested an Apache server containing a vulnerable OpenSSL implementation (before 0.9.6e) that contained a buffer overflow vulnerability that a remote attacker could exploit to inject code [13]. When instruction set tagging is used, the exploit is disrupted since it does not contain the proper instruction tags in the injected code.

We also conducted some performance measurements on our 2-variant system with memory address partitioning. The average response latency for HTTP requests increased from 0.2ms for the unmodified server to 2.9ms for the 2-variant system.

The proof-of-concept implementation validated the N-variant systems framework concept, but did not provide a practical or secure implementation for realistic services. Due to isolation of the variants, various non-attack inputs could lead to divergences between the variants caused by differences between the hosts. For example, if the output web page includes a time stamp or host IP address, these would differ between the variants. This means false positives could occur when the monitor observes differences between the outputs for normal requests. Furthermore, a motivated attacker could take advantage of any of these differences to construct an attack that would compromise one of the variants without leading to a detected divergence.

5.2 Kernel Implementation

The difficulties in eliminating nondeterminism and providing finer grain monitoring with the isolated implementation, as well as its performance results, convinced us to develop a kernel implementation of the framework by modifying the Linux 2.6.11 kernel. In this implementation, all the variants run on the same platform, along with the polygrapher and monitor. We rely on existing operating system mechanisms to provide isolation between the variants, which execute as separate processes.

We modified the kernel data structures to keep track of variant processes and implemented wrappers around system calls. These wrappers implement the polygraphing functionality by wrapping input system calls so that when both variants make the same input system call, the actual input operation is performed once and the same data is sent to all variants. They provide the monitoring functionality by checking that all variants make the same call with equivalent arguments before making the actual system call.

This system call sharing approach removes nearly all of the causes of nondeterminism that were problematic in the proof-of-concept implementation. By wrapping the system calls, we ensure that variants receive identical results from all system calls. The remaining cause of nondeterminism is due to scheduling differences, in particular in handling signals. We discuss these limitations in Section 6.

In order to bring an N-variant system into execution we created two new system calls: n_variant_fork, and n_variant_execve. The program uses these system calls similarly to the way a shell uses fork/execve to bring processes into execution. The n_variant_fork system call forks off the variants, however instead of creating a single child process it creates one process per variant. The variants then proceed to call n_variant_execve, which will cause each of the variants to execute their own diversified binary of the server. Note that our approach requires no modification of an existing binary to execute it within an N-variant system; we simply invoke a shell command that takes the pathnames of variant binaries as parameters and executes n_variant_execve.

Next, we provide details on the system call wrappers that implement the polygraphing and monitoring. The

```
ssize_t sys_read(int fd, const void *buf, size_t count) {
  if (!hasSibling (current)) { make system call normally } // not a variant process
  else {
    record that this variant process entered call
    if (!inSystemCall (current->sibling)) { // this variant is first
      save parameters
      sleep // sibling will wake us up
      get result and copy *buf data back into address space
      return result;
    } else if (currentSystemCall (current->sibling) == SYS_READ) { // this variant is second, sibling waiting
      if (parameters match) { // what it means to "match" depends on variation and system call
        perform system call
        save result and data in kernel buffer
        wake up sibling
        return result;
      } else { DIVERGENCE ERROR! } // sibling used different parameters
    } else { DIVERGENCE ERROR! } } } // sibling is in a different system call
```

Figure 2. Typical shared system call wrapper.

Linux 2.6.11 kernel provides 267 system calls. We generalize them into three categories based on the type of wrapper they need: shared system calls, reflective system calls, and dangerous system calls.

Shared System Calls. For system calls that interact with external state, including I/O system calls, the wrapper checks that all variants make equivalent calls, makes the actual call once, and sends the output to all variants, copying data into each of the variants address space if necessary. Figure 2 shows pseudocode for a shared call, in this case the read system call. The actual wrappers are generated using a set of preprocessor macros we developed to avoid duplicating code. The first if statement checks whether this process is part of an N-variant system. If not, the system call proceeds normally. Hence, a single platform can run both normal and N-variant processes. If the process is a variant process, it records that it has entered this system call and checks if its sibling variant has already entered a system call. If it has not, it saves the parameters and sleeps until the other variant wakes it up. Otherwise, it checks that the system call and its parameters match those used by the first variant to make the system call. If they match, the actual system call is made. The result is copied into a kernel buffer, and the sibling variant process (which reached this system call first and went to sleep) is awoken. The sibling process copies the result from the kernel buffer back into its address space and continues execution.

Reflective System Calls. We consider any system call that observes or modifies properties of the process itself a *reflective* system call. For these calls, we need to ensure that all observations always return the same value regardless of which variant reaches the call first, and that all modifications to process properties are done equivalently on all variants. For observation-only reflective calls, such as getpid, we check that all variants make the same call, and then just make the call once for variant 0 and send the same result to all variants. This is done using wrappers similar to those for shared system calls, except instead of just allowing the last variant that reaches the call to make the actual system call we need to make sure that each time a reflective call is reached, it is executed for the same process.

Another issue is raised by the system calls that create child processes (sys_fork, sys_vfork, and sys_clone). The wrappers for these calls must coordinate each variant's fork and set up all the child processes as a child N-variant system before any of the children are placed on the run queue. These system calls return the child process' PID. We ensure that all the parents in the N-variant system get the same PID (the PID of variant 0's child), as with the process observation system calls.

The other type of reflective system call acts on the process itself. These system calls often take parameters given by the reflective observation system calls. In this case, we make sure they make the same call with the same parameters, but alter the parameters accordingly for each variant. For example, sys_wait4 takes a PID as an input. Each of the variants will call sys_wait4 with the same PID because they were all given the same child PID when they called sys_fork (as was required to maintain normal equivalence). However, each variant needs to clean up its corresponding child process within the child system. The wrapper for sys_wait4 modifies the PID value passed in and makes the appropriate call for each variant with its corresponding child PID. Similar issues arise with sys_kill, sys_tkill, and sys_waitpid.

Finally, we have to deal with two system calls that terminate a process: sys_exit and sys_exit_group. A terminating process does not necessarily go through these system calls, since it may terminate by crashing. To ensure that we capture all process termination events in an N-variant system we added a monitor inside the do_exit function within the kernel which is the last function all terminating processes execute. This way, if a process receives a signal and exits without going through a system call, we will still observe this and can terminate the other variants.

Dangerous System Calls. Certain calls would allow processes to break assumptions on which we rely. For example, if the process uses the execve system to run a new executable, this will escape the N-variant protections unless we can ensure that each variant executes a different executable that is diversified appropriately. Since it is unlikely we can establish this property, the execve wrapper just disables the system call and returns an error code. This did not pose problems for Apache, but might for other applications.

Other examples of dangerous system calls are those for memory mapping (old_mmap, sys_mmap2) which map a portion of a file into a process' address space. After a file is mapped into an address space, memory reads and writes are analogous to reads and writes from the file. This would allow an attacker to compromise one variant, and then use the compromised variant to alter the state of the uncompromised variants through the shared memory without detection, since no system call is necessary. Since many server applications (including

Configuration		1	2	3	4	5	6
Description		Unmodified Apache, unmodified kernel	Unmodified Apache, N-variant kernel	2-variant system, address partitioning	Apache running under Strata	Apache with instruction tags	2-variant system, instruction tags
Unsaturated	Throughput (MB/s)	2.36	2.32	2.04	2.27	2.25	1.80
	Latency (ms)	2.35	2.40	2.77	2.42	2.46	3.02
Saturated	Throughput (MB/s)	9.70	9.59	5.06	8.54	8.30	3.55
	Latency (ms)	17.65	17.80	34.20	20.30	20.58	48.30

Apache) use memory mapping, simply blocking these system calls is not an option. Instead, we place restrictions on them to allow only the MAP_ANONYMOUS and MAP_PRIVATE options with all permissions and to permit MAP_SHARED mappings as long as write permissions are not requested. This eliminates the communication channel between the variants, allowing memory mapping to be used safely by the variants. Apache runs even with these restrictions since it does not use other forms of memory mapping, but other solutions would be needed to support all services.

5.3 Performance

Table 1 summarizes our performance results. We measured the throughput and latency of our system using WebBench 5.0 [65], a web server benchmark using a variety of static web page requests. We ran two sets of experiments measuring the performance of our Apache server under unsaturated and saturated load conditions. In both sets, there was a single 2.2GHz Pentium 4 server machine with 1GB RAM running Fedora Core 3 (2.6.11 kernel) in the six different configurations shown in Table 1. For the first set of experiences, we used a single client machine running one WebBench client engine. For the load experiments, we saturated our server using six clients each running five WebBench client engines connected to the same networks switch as the server.

Configuration 1 is the baseline configuration: regular apache running on an unmodified kernel. Configuration 2 shows the overhead of the N-variant kernel on a normal process. In our experiments, it was negligible; this is unsurprising since the overhead is only a simple comparison at the beginning of each wrapped system call. Configuration 3 is a 2-variant system running in our N-variant framework where the two variants differ in the address spaces according to the partitioning scheme described in Section 4.1. For the unloaded server, the latency observed by the client increases by 17.6%. For the loaded server, the throughput decreases by 48% and the latency nearly doubles compared to the baseline configuration. Since the N-variant system executes all computation twice, but all I/O system calls only once, the overhead incurred reflects the cost of duplicating the computation, as well as the checking done by the wrappers. The overhead measured for the unloaded server is fairly low, since the process is primarily I/O bound; for the loaded server, the process becomes more compute-bound, and the approximately halving of throughput reflects the redundant computation required to run two variants.

The instruction tagging variation is more expensive because of the added cost of removing and checking the instruction tags. Configuration 4 shows the performance of Apache running on the normal kernel under Strata with no transformation. The overhead imposed by Strata reduces throughput by about 10%. The Strata overhead is relatively low because once a code fragment is in the fragment cache it does not need to be translated again the next time it executes. Adding the instruction tagging (Configuration 5) has minimal impact on throughput and latency. Configuration 6 shows the performance of a 2-variant system where the variants are running under Strata with instruction tag variation. The performance impact is more than it was in Configuration 3 because of the additional CPU workload imposed by the instruction tags. For the unloaded server, the latency increases 28% over the baseline configuration; for the saturated server, the throughput is 37% of the unmodified server's throughput.

Our results indicate that for I/O bound services, N-variant systems where the variation can be achieved with reasonable performance overhead, especially for variations such as the address space partitioning where little additional work is needed at run-time. We anticipate there being many other interesting variations of this type, such as file renaming, local memory rearrangement, system call number diversity, and user

id diversity. For CPU-bound services, the overhead of our approach will remain relatively high since all computation needs to be performed twice. Multiprocessors may alleviate some of the problem (in cases where there is not enough load to keep the other processors busy normally). Fortunately, many important services are largely I/O-bound today and trends in processor and disk performance make this increasingly likely in the future.

6. Discussion

Our prototype implementation illustrates the potential for N-variant systems to protect vulnerable servers from important classes of attacks. Many other issues remain to be explored, including how our approach can be applied to other services, what variations can be created to detect other classes of attacks, how an N-variant system can recover from a detected attack, and how compositions of design and artificially diversified variants can provide additional security properties.

Applicability. Our prototype kernel implementation demonstrated the effectiveness of our approach using Apache as a target application. Although Apache is a representative server, there are a number of things other servers might do that would cause problems for our implementation. The version of Apache used in our experiments on uses the fork system call to create separate processes to handle requests. Each child process is run as an independent N-variant system. Some servers use user-level threading libraries where there are multiple threads within a single process invisible to our kernel monitor. This causes problems in an N-variant system, since the threads in the variants may interleave differently to produce different sequences of system calls (resulting in a false detection), or worse, interleave in a way that allows an attacker to exploit a race condition to carry out a successful attack without detection. One possible solution to this problem is to modify the thread scheduler to ensure that threads in the variants are scheduled identically to preserve synchronization between the variants.

The asynchronous property of process signals makes it difficult to ensure that all variants receive a signal at the exact same point in each of their executions. Although we can ensure that a signal is sent to all the variants at the same time, we cannot ensure that all the variants are exactly at the same point within their program at that time. As a result, the timing of a particular signal could cause divergent behavior in the variants if the code behaves differently depending on the exact point when the signal is received. This might cause the variants to diverge even though they are not under attack, leading to a false positive detection. As with user-level threads, if we modify the kernel to provide more control of the scheduler we could ensure that variants receive signals at the same execution points.

Another issue that limits application of our approach is the use of system calls we classified as dangerous such as execve or unrestricted use of mmap. With our current wrappers, a process that uses these calls is terminated since we cannot handle them safely in the N-variant framework. In some cases, more precise wrappers may allow these dangerous calls to be used safely in an N-variant system. Some calls, however, are inherently dangerous since they either break isolation between the variants or allow them to escape the framework. In these situations, either some loss of security would need to be accepted, or the application would need to be modified to avoid the dangerous system calls before it could be run as an N-variant system.

Other variations. The variations we have implemented only thwart attacks that require accessing absolute memory addresses or injecting code. For example, our current instruction tagging variation does not disrupt a *return-to-libc* attack (since it does not involve injecting code), and our address space partitioning variation provides no protection against memory corruption attacks that only use relative addressing. One goal for our future work is to devise variations that enable detection of larger classes of attack within the framework we have developed. We believe there are rich opportunities for incorporating different kinds of variation in our framework, although the variants must be designed carefully to ensure the detection and normal equivalence properties are satisfied. Possibilities include variations involving memory layout to prevent classes of relative addressing attacks, file system paths to disrupt attacks that depend on file names, scheduling to thwart race condition attacks, and data structure parameters to disrupt algorithmic complexity attacks [21].

Composition. Because of the need to satisfy the normal equivalence property, we cannot simply combine multiple variations into two variants to detect the union of their attack classes. In fact, such a combination risks compromising the security properties each variation would provide by itself. By combining variations more carefully, however, we can compose variants in a way that maintains the properties of the independent variations. To do this securely, we must ensure that, for

each attack class we wish to detect, there is a pair of variants in the system that differs only in the transformation used to detect that attack class. This is necessary to ensure that for each variation, there is a pair of variants that satisfy the normal equivalence property for that variation but differ in the varied property. This approach can generalize to compose *n* binary variations using *n* + 1 variants. More clever approaches may be able to establish the orthogonality of certain variations to allow fewer variants without sacrificing normal equivalence.

Another promising direction is to combine our approach with design diversity approaches [46, 28, 62]. We could create a 3-variant system where two variants are Apache processes running on Linux hosts with controlled address space partitioning variation, and the third variant is a Windows machine running IIS. This would provide guaranteed detection of a class of low-level memory attacks through the two controlled variants, as well as probabilistic detection of attacks that exploit high-level application semantics through the design variants.

Recovery. Our modified kernel detects an attack when the system calls made by the variants diverge. At this point, one variant is in an alarm state (e.g., crashed), and the other variant is in a possibly compromised state. After detecting the attack, the monitor needs to restart the service in an uncompromised state. Note that the attack is always detected before any system call is executed for a compromised process; this means no external state has been corrupted. For a stateless server, the monitor can just restart all of the variants. For a stateful server, recovery is more difficult. One interesting approach is to compare the states of the variants after the attack is detected to determine the valid state. Depending on the variation used, it may be possible to recover a known uncompromised state from the state of the alarm variant, as well as to deduce an attack signature from the differences between the two variants' states. Another approach involves adding an extra *recovery variant* that maintains a known uncompromised state and can be used to restart the other variants after an attack is detected. The recovery variant could be the original *P*, except it would be kept behind the normal variants. The polygrapher would delay sending input to the recovery variant until all of the regular variants process it successfully. This complicates the wrappers substantially, however, and raises difficult questions about how far behind the recovery variant should be.

7. Conclusion

Although the cryptography community has developed techniques for proving security properties of cryptographic protocols, similar levels of assurance for system security properties remains an elusive goal. System software is typically too complex to prove it has no vulnerabilities, even for small, well-defined classes of vulnerabilities such as buffer overflows. Previous techniques for thwarting exploits of vulnerabilities have used ad hoc arguments and tests to support claimed security properties. Motivated attackers, however, regularly find ways to successfully attack systems protected using these techniques [12, 55, 58, 64].

Although many defenses are available for the particular attacks we address in this paper, the N-variant systems approach offers the promise of a more formal security argument against large attack classes and correspondingly higher levels of assurance. If we can prove that the automated diversity produces variants that satisfy both the normal equivalence and detection properties against a particular attack class, we can have a high degree of confidence that attacks in that class will be detected. The soundness of the argument depends on correct behavior of the polygrapher, monitor, variant generator and any common resources.

Our framework opens up exciting new opportunities for diversification approaches, since it eliminates the need for high entropy variations. By removing the reliance on keeping secrets and providing an architectural and associated proof framework for establishing security properties, N-variant systems offer potentially substantial gains in security for high assurance services.

Availability

Our implementation is available as source code from http://www.nvariant.org. This website also provides details on the different system call wrappers.

Acknowledgments

We thank Anil Somayaji for extensive comments and suggestions; Lorenzo Cavallaro for help with the memory partitioning scripts; Jessica Greer for assistance setting up our experimental infrastructure; Caroline Cox, Karsten Nohl, Nate Paul, Jeff Shirley, Nora Sovarel, Sean Talts, and Jinlin Yang for comments on the work and writing. This work was supported in part by grants from the DARPA Self-Regenerative Systems Program (FA8750-04-2-0246) and the National Science Foundation through NSF Cybertrust (CNS-0524432).

References

[1] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. *CCS* 2005.

[2] Starr Andersen. Changes to Functionality in Microsoft Windows XP Service Pack 2: Part 3: Memory Protection Technologies. *Microsoft TechNet*. August 2004.

[3] Algirdas Avizienis and L. Chen. On the Implementation of N-version Programming for Software Fault-Tolerance During Program Execution. *International Computer Software and Applications Conference*. 1977.

[4] Vasanth Bala, E. Duesterwald, S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. *ACM Programming Language Design and Implementation* (PLDI). 2000.

[5] Arash Baratloo, N. Singh, T. Tsai. Transparent Run-Time Defense against Stack Smashing Attacks. *USENIX Technical Conference*. 2000.

[6] Elena Barrantes, D. Ackley, S. Forrest, T. Palmer, D. Stefanovic, D. Zovi. Intrusion Detection: Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. *CCS* 2003.

[7] Emery Berger and Benjamin Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. *ACM Programming Language Design and Implementation* (PLDI), June 2006.

[8] Sandeep Bhatkar, Daniel DuVarney, and R. Sekar. Address Ofuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. *USENIX Security* 2003.

[9] Kenneth Birman. Replication and Fault Tolerance in the ISIS System. *10th ACM Symposium on Operating Systems Principles*, 1985.

[10] K. Birman, *Building Secure and Reliable Network Applications*, Manning Publications, 1996.

[11] Derek Bruening, Timothy Garnett, Saman Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. *International Symposium on Code Generation and Optimization*. 2003.

[12] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack*. Vol 0xa Issue 0x38. May 2000. http://www.phrack.org/phrack/56/p56-0x05

[13] CERT. *OpenSSL Servers Contain a Buffer Overflow During the SSL2 Handshake Process*. CERT Advisory CA-2002-23. July 2002.

[14] L. Chen and Algirdas Avizienis. N-Version Programming: A Fault Tolerance Approach to Reliability of Software Operation. *8th International Symposium on Fault-Tolerant Computing*. 1978.

[15] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, R. K. Iyer. Non-Control-Data Attacks Are Realistic Threats. *USENIX Security* 2005.

[16] Marc Chérèque, David Powell, Philippe Reynier, Jean-Luc Richier, and Jacques Voiron. Active Replication in Delta-4. *22nd International Symposium on Fault-Tolerant Computing*. July 1992.

[17] Monica Chew and Dawn Song. *Mitigating Buffer Overflows by Operating System Randomization*. Tech Report CMU-CS-02-197. December 2002.

[18] George Coulouris, Jean Dollimore and Tim Kindberg. *Distributed Systems: Concepts and Design* (Third Edition). Addison-Wesley. 2001.

[19] Crispin 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. *USENIX Security* 1998.

[20] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. *USENIX Security* 2001.

[21] Scott Crosby and Dan Wallach. Denial of Service via Algorithmic Complexity Attacks. *USENIX Security* 2003.

[22] Bruno De Bus, Bjorn De Sutter, Ludo Van Put, D. Chanet, K. De Bosschere. Link-time Optimization of ARM Binaries. Language. *Compiler and Tool Support for Embedded Systems*. 2004.

[23] Nurit Dor, M. Rodeh, M. Sagiv. CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. *ACM Programming Language Design and Implementation*. June 2003.

[24] Jon Erickson**.** *Hacking: The Art of Exploitation*. No Starch Press. November 2003,

[25] Hiroaki Etoh. *GCC extension for protecting applications from stack-smashing attacks*. IBM, 2004. http://www.trl.ibm.com/projects/security/ssp

[26] Stephanie Forrest, Anil Somayaji, David Ackley. Building diverse computer systems. *6th Workshop on Hot Topics in Operating Systems*. 1997.

[27] The FreeBSD Documentation Project. *FreeBSD Handbook*, Chapter 24. 2005.

[28] Debin Gao, Michael Reiter, Dawn Song**.** Behavioral Distance for Intrusion Detection. *8th International Symposium on Recent Advances in Intrusion Detection*. September 2005.

[29] Daniel Geer, C. Pfleeger, B. Schneier, J. Quarterman, P. Metzger, R. Bace, P. Gutmann. *Cyberinsecurity: The Cost of Monopoly*. CCIA Technical Report, 2003.

[30] Eric Haugh and Matt Bishop. Testing C programs for buffer overflow vulnerabilities. *NDSS* 2003.

[31] David Holland, Ada Lim, and Margo Seltzer. An Architecture A Day Keeps the Hacker Away.

Workshop on Architectural Support for Security and Anti-Virus. April 2004.

[32] D. Jewett. Integrity S2: A Fault-Tolerant Unix Platform. *17th International Symposium on Fault-Tolerant Computing Systems*. June 1991.

[33] Mark K. Joseph. *Architectural Issues in Fault-Tolerant, Secure Computing Systems*. Ph.D. Dissertation. UCLA Department of Computer Science, 1988.

[34] James Just, J. Reynolds, L. Clough, M. Danforth, K. Levitt, R. Maglich, J. Rowe. Learning Unknown Attacks – A Start. *Recent Advances in Intrusion Detection*. Oct 2002.

[35] Gaurav Kc, A. Keromytis, V. Prevelakis. Countering Code-injection Attacks with Instruction Set Randomization. *CCS* 2003.

[36] John Knight and N. Leveson. An Experimental Evaluation of the Assumption of Independence in Multi-version Programming. *IEEE Transactions on Software Engineering*, Vol 12, No 1. Jan 1986.

[37] Ken Knowlton. A Combination Hardware-Software Debugging System. *IEEE Transactions on Computers*. Vol 17, No 1. January 1968.

[38] Benjamin Kuperman, C. Brodley, H. Ozdoganoglu, T. Vijaykumar, A. Jalote. Detection and Prevention of Stack Buffer Overflow Attacks. *Communications of the ACM*, Nov 2005.

[39] David Larochelle and David Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. *USENIX Security* 2001.

[40] Ruby Lee, D. Karig, J. McGregor, and Z. Shi. Enlisting Hardware Architecture to Thwart Malicious Code Injection. *International Conference on Security in Pervasive Computing*. March 2003.

[41] John McGregor, David Karig, Zhijie Shi, and Ruby Lee. A Processor Architecture Defense against Buffer Overflow Attacks. *IEEE International Conference on Information Technology: Research and Education*. August 2003.

[42] Sjoerd Mullender and Robbert van Renesse. The International Obfuscated C Code Contest Entry. 1984. http://www1.us.ioccc.org/1984/mullender.c

[43] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. *NDSS* 2005.

[44] Adam J. O'Donnell and H. Sethu. On Achieving Software Diversity for Improved Network Security using Distributed Coloring Algorithms. *CCS* 2004.

[45] Manish Prasad and T. Chiueh. A Binary Rewriting Defense against Stack-Based Buffer Overflow Attacks. *USENIX Technical Conference*. June 2003.

[46] James Reynolds, J. Just, E. Lawson, L. Clough, R. Maglich, K. Levitt. The Design and Implementation of an Intrusion Tolerant System. *Foundations of Intrusion Tolerant Systems* (OASIS). 2003.

[47] Michael Ringenburg and Dan Grossman. Preventing Format-String Attacks via Automatic and Efficient Dynamic Checking. *CCS* 2005.

[48] Juan Rivas. *Overwriting the .dtors Section*. Dec 2000. http://synnergy.net/downloads/papers/dtors.txt

[49] Olatunji Ruwase and Monica S. Lam. A Practical Dynamic Buffer Overflow Detector. *NDSS* 2004.

[50] Fred Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*. Dec 1990.

[51] Fred Schneider and L. Zhou. *Distributed Trust: Supporting Fault-Tolerance and Attack-Tolerance*, Cornell TR 2004-1924, January 2004.

[52] Kevin Scott and Jack W. Davidson. Safe Virtual Execution Using Software Dynamic Translation. *ACSAC*. December 2002.

[53] Kevin Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, M. L. Soffa. Retargetable and Reconfigurable Software Dynamic Translation. *International Symposium on Code Generation and Optimization*. March 2003.

[54] Scut / team teso. *Exploiting Format String Vulnerabilities*. March 2001.

[55] Hovav Shacham, M. Page, B. Pfaff, Eu-Jin Goh, N. Modadugu, Dan Boneh. On the effectiveness of address-space randomization. *CCS* 2004.

[56] Umesh Shankar, K. Talwar, J. Foster, D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. *USENIX Security* 2001.

[57] Stelios Sidiroglou, G. Giovanidis, A. Keromytis. A Dynamic Mechanism for Recovering from Buffer Overflow Attacks. *8th Information Security Conference*. September 2005.

[58] Ana Nora Sovarel, David Evans, Nathanael Paul. Where's the FEEB?: The Effectiveness of Instruction Set Randomization. *USENIX Security* 2005.

[59] Mark Stamp. Risks of Monoculture. *Communications of the ACM*. Vol 47, Number 3. March 2004.

[60] Karthik Sundaramoorthy, Z. Purser, E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. *Architectural Support for Programming Languages and Operating Systems* (ASPLOS). Nov 2000.

[61] Bjorn De Sutter and Koen De Bosschere. Introduction: Software techniques for Program

Compaction. *Communications of the ACM*. Vol 46, No 8. Aug 2003.

[62] Eric Totel, Frédéric Majorczyk, Ludovic Mé. COTS Diversity Intrusion Detection and Application to Web Servers. *Recent Advances in Intrusion Detection*. September 2005.

[63] Timothy Tsai and Navjot Singh. *Libsafe 2.0: Detection of Format String Vulnerability Exploits*. Avaya Labs White Paper. February 2001.

[64] Nathan Tuck, B. Calder, and G. Varghese. Hardware and Binary Modification Support for Code Pointer Protection from Buffer Overflow**.** *International Symposium on Microarchitecture*. Dec 2004.

[65] VeriTest Corporation. *WebBench 5.0*. http://www.veritest.com/benchmarks/webbench

[66] John Viega, J. Bloch, T. Kohno, Gary McGraw. ITS4 : A Static Vulnerability Scanner for C and C++ Code. *ACSAC*. Dec 2000.

[67] David Wagner, J. Foster, E. Brewer, A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. *NDSS* 2000.

[68] D. Wilson. The STRATUS Computer System. *Resilient Computer Systems: Volume 1*. John Wiley and Sons, 1986. p. 208-231.

[69] Jun Xu, Z. Kalbarczyk, R. Iyer. Transparent Runtime Randomization for Security. *Symposium on Reliable and Distributed Systems*. October 2003.

[70] Yongguang Zhang, H. Vin, L. Alvisi, W. Lee, S. Dao. Heterogeneous Networking: a New Survivability Paradigm. *New Security Paradigms Workshop* 2001.

Appendix F: PHPrevent – Web Application Security

Automatically hardening web applications using precise tainting

Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, David Evans
Department of Computer Science, University of Virginia, 151 Engineer's Way, Charlottesville, VA 22904-4740, USA [⊥]

Abstract: Most web applications contain security vulnerabilities. The simple and natural ways of creating a web application are prone to SQL injection attacks and cross-site scripting attacks as well as other less common vulnerabilities. In response, many tools have been developed for detecting or mitigating common web application vulnerabilities. Existing techniques either require effort from the site developer or are prone to false positives. This paper presents a fully automated approach to securely hardening web applications. It is based on precisely tracking taintedness of data and checking specifically for dangerous content only in parts of commands and output that came from untrustworthy sources. Unlike previous work in which everything that is derived from tainted input is tainted, our approach precisely tracks taintedness within data values.

Key words: web security; web vulnerabilities; SQL injection; PHP; cross-site scripting attacks; precise tainting; information flow

1. Introduction

Nearly all web applications are security critical, but only a small fraction of deployed web applications can afford a detailed security review. Even when such a review is possible, it is tedious and can overlook subtle security vulnerabilities. Serious security vulnerabilities are regularly found in the most prominent commercial web applications including Gmail[1], eBay[2], Yahoo[3], Hotmail[3] and Microsoft Passport[4]. Section 2 provides background on common web application vulnerabilities.

Several tools have been developed to partially automate aspects of a security review, including static analysis tools that scan code for possible vulnerabilities[5] and automated testing

[⊥] This work was funded in part by DARPA (SRS FA8750-04-2-0246) and the National Science Foundation (NSF CAREER CCR-0092945, SCI-0426972).

tools that test web sites with inputs designed to expose vulnerabilities[5-7]. Taint analysis identifies inputs that come from untrustworthy sources (including user input) and tracks all data that is affected by those input values. An error is reported if tainted data is passed as a security-critical parameter, such as the command passed to an exec command. Taint analysis can be done statically or dynamically. Section 3 describes previous work on securing web applications, including taint analysis.

For an approach to be effective for the vast majority of web applications, it needs to be fully automated. Many people build websites that accept user input without any understanding of security issues. For example, *PHP & MySQL for Dummies*[8] provides inexperienced programmers with the knowledge they need to set up a database-backed web application. Although the book does include some warnings about security (for example, p. 213 warns readers about malicious input and advises them to check correct format, and p. 261 warns about <script> tags in user input), many of the examples in the book that accept user input contain security vulnerabilities (e.g., Listings 11-3 and 12-2 allow SQL injection, and Listing 12-4 allows cross-site scripting). This is typical of most introductory books on web site development.

In Section 4 we propose a completely automated mechanism for preventing two important classes of web application security vulnerabilities: command injection (including script and SQL injection) and cross-site scripting (XSS). Our solution involves replacing the standard PHP interpreter with a modified interpreter that precisely tracks taintedness and checks for dangerous content in uses of tainted data. All that is required to benefit from our approach is that the hosting server uses our modified version of PHP.

The main contribution of our work is the development of precise tainting in which taint information is maintained at a fine level of granularity and checked in a context-sensitive way. This enables us to design and implement fully-automated defense mechanisms against both command injection attacks, including SQL injection, and cross-site scripting attacks. Next, we describe common web application vulnerabilities. Section 3 reviews prior work on securing web applications. Section 4 describes our design and implementation, and explains how we prevent exploits of web application vulnerabilities.

2. Web Application Vulnerabilities

Figure 1 depicts a typical web application. For clarity, we focus on web applications implemented using PHP, which is currently one of the most popular language for implementing web applications (PHP is used at approximately 1.3M IP addresses, 18M domains, and is installed on 50% of Apache servers[9]). Most issues and architectural properties are similar for other web application languages.

A client sends input to the web server in the form of an HTTP request (step 1 in Figure 1). `GET` and `POST` are the most common requests. The request encodes data created by the user in HTTP header fields including file names and parameters included in the requested URI. If the

URI is a PHP file, the HTTP server will load the requested file from the file system (step 2) and execute the requested file in the PHP interpreter (step 3). The parameters are visible to the PHP code through predefined global variable arrays (including $_GET and $_POST).

The PHP code may use these values to construct commands that are sent to PHP functions such as a SQL query that is sent to the database (steps 4 and 5), or to make calls to PHP API functions that call system APIs to manipulate system state (steps 6 and 7). The PHP code produces an output web page based on the returned results and returns it to the client (step 8).

We assume a client can interact with the web server only by sending HTTP requests to the HTTP server. In particular, the only way an attacker can interact with system resources, including the database and file system, is by constructing appropriate web requests. We divide attacks into two general classes of attacks: *injection attacks* attempt to construct requests to the web server that corrupt its state or reveal confidential information; *output attacks* (e.g., cross-site scripting) attempt to send requests to the web server that cause it to generate responses that produce malicious behavior on clients.

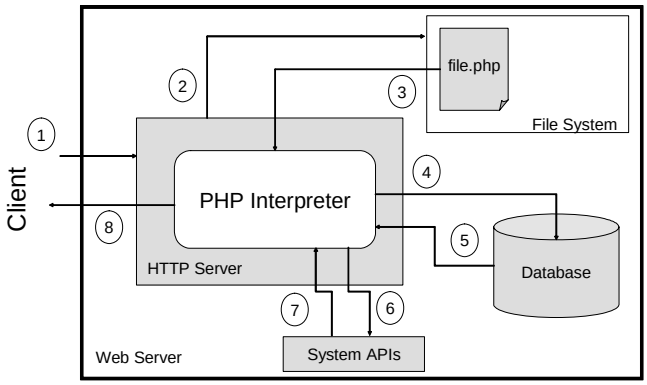


Figure 1. *Typical web application architecture*

a. Command injection attacks

In a command injection attack an attacker attempts to access confidential information or corrupt the application state by constructing an input that allows the attacker to inject malicious control logic into the web application. With the system architecture shown in Figure 1, an attack could attempt to inject PHP code that will be executed by the PHP interpreter, SQL commands that will be executed by the database, or native machine code that will be executed by the web server host directly. We consider only the first two cases. Web application vulnerabilities are far more common than vulnerabilities in the underlying server or operating system since there are

far more different web applications than there are servers and operating systems, and developers of web applications tend to be far less sophisticated from a security perspective than developers of operating systems and web servers.

PHP injection. In a PHP injection attack, the attacker attempts to inject PHP code that will be interpreter by the server. If an attacker can inject arbitrary code, the attacker can do everything PHP can and has effectively complete control over the server. Here is a simple example of a PHP injection in phpGedView, an online viewing system for genealogy information[10]. The attack URL is of the form:

```
http://[target]/[...]/editconfig_gedcom.php?gedcom_config=../../../../../../etc/passwd
```

The vulnerable PHP code uses the gedcom_config value as a filename: require($gedcom_config);. The semantics of require is to load the file and either interpret it as PHP code (if the PHP tags are found) or display the content. Thus this code leaks the content of the password file. Abuse of require and its related functions is a commonly reported occurrence[11,12], despite the fact that, properly configured, PHP is impervious to this basic attack. However, additional defenses are needed for more sophisticated injection attacks such as the recently released Santy Worm[13] and the phpMyAdmin attack[14].

SQL injection. Attacking web applications by injecting SQL commands is a common method of attacking web applications[15,16]. We illustrate a simple SQL injection that is representative of actual vulnerabilities. Suppose the following is used to construct an SQL query to authenticate users against a database:

```
$cmd="SELECT user FROM users WHERE user = ' " . $user
        . '" AND password = ' " . $pwd . " ' ";
```

The value of $user comes from $_POST['user'], a value provided by the client using the login form. A malicious client can enter the value: ' OR 1 = 1 ; --' (-- begins a comment in SQL which continues to the end of the line). The resulting SQL query will be: SELECT user FROM users WHERE user = ' ' OR 1 = 1 ; -- ' AND password = 'x'. The injected command closes the quote and comments out the AND part of the query. Hence, it will always succeed regardless of the entered password.

The main problem here is that the single quote provided by the attacker closes the open quote, and the remainder of the user-provided string is passed to the database as part of the SQL command. This attack would be thwarted by PHP installations that use the default magic quotes option. When enabled, magic quotes automatically sanitize input data by adding a backslash to all strings submitted via web forms or cookies. However, magic quotes do not suffice for attacks that do not use quotes[17].

One solution to prevent SQL injections is to use prepared statements[18]. A prepared statement is a query string with placeholders for variables that are subsequently bound to the statement and type-checked. However, this depends on programmers changing development practices and replacing legacy code. Dynamic generation of queries using regular queries will continue to be prevalent for the foreseeable future.

b. Output attacks

Output attacks send a request to a web application that causes it to produce an output page designed by the attacker to achieve some malicious goal. The most dangerous kind of output attack is a *cross-site scripting* attack, in which the web server produces an output page containing script code generated by the attacker. The script code can steal the victim's cookies or capture data the victim unsuspectingly enters into the web site. This is especially effective in phishing attacks in which the attacker sends potential victims emails convincing them victim to visit a URL. The URL may be a trusted domain, but because of a cross-site scripting vulnerability the attacker can construct parameters to the URL that cause the trusted site to create a page containing a form that sends data back to the attacker. For example, the attacker constructs a link like this:

```
<a href='http://bad.com/go.php?val=<script src="http://bad.com/attack.js"></script>'>
```

If the implementation of `go.php` uses the `val` parameter in the generated web page output (for example, by doing `print "Results for: " . $_GET['val'];`), the malicious script will appear on the resulting page. A clever attacker can use character encodings to make the malicious script appear nonsensical to a victim who inspects the URL before opening it.

Five years ago, CERT Advisory 2000-02 described the problem of cross-site scripting and advised users to disable scripting languages and web site developers to validate web page output[19]. Nevertheless, cross-site scripting problems remain a serious problem today. Far too much functionality of the web depends on scripting languages, so most users are unwilling to disable them. Even security-conscious web developers frequently produce websites that are vulnerable to cross-site scripting attacks[1,4,20-22]. As with SQL injection, ad hoc fixes often fail to solve discovered problems correctly—the initial filters develop to fix the Hotmail vulnerability could be circumvented by using alternate character encodings[4]. Hence, we focus on fully automated solutions.

3. Related work

Several approaches have been developed for securing web applications including filtering input and output that appears dangerous, automated testing and diversity defenses. The approaches most similar to our proposed approach involve analyzing information flow.

Input and Output Filtering. Scott and Sharp developed a system for providing an application-level firewall to prevent malicious input from reaching vulnerable web servers[23]. Their approach required a specification of constraints on different inputs, and compiled those constraints into a checking program. This requires a programmer to provide a correct security policy specific to their application, so is ill-suited to protecting typical web developers. Several

commercial web application firewalls provide input and output filtering to detect possible attacks[24,25]. However, these tools are prone to both false positives and negatives[26].

Automated Testing. There are several web application security testing tools designed specifically to find vulnerabilities[5,27,28]. The problem with these tools is that they have to guess the exploit data in order to expose the vulnerability. For well-known generic classes of vulnerabilities, such as SQL injection, this may be possible. But for novel or complex vulnerabilities, it is unlikely the scanner will guess the right inputs to expose the vulnerability.

Diversity Defenses. Instruction-Set Randomization is a form of diversity in which defenders modify the instruction set used to run applications[29]. Thus, code-injection attacks that rely on knowledge of the original language are detected and thwarted easily. This approach has been advocated for general scripting languages[29] and for protection against SQL injections[30]. There are two main problems with ISR: (1) it is effective only against code injection attacks and incomplete by itself (it does not handle cross-site scripting attacks), and (2), the deployment of ISR is not transparent to developers and requires the transformation of application code.

Information Flow. All of the web vulnerabilities described in Section 0 stem from insecure information flow: data from untrusted sources is used in a trusted way. The security community has studied information flow extensively[31]. The earliest work focused on confidentiality, in particular in preventing flows from trusted to untrusted sources[32]. In our case, we are primarily concerned with integrity. Biba showed that information flow can also be used to provide integrity by considering flows from untrusted to trusted sources[33].

Information flow policies can be enforced statically, dynamically or by a combination of static and dynamic techniques. Static taint analysis has been used to detect security vulnerabilities in C programs[34,35]. Static approaches have the advantage of increased precision, no run-time overhead and the ability to detect and correct errors before deployment. However, they require substantial effort from the programmer. Since we are focused on solutions that will be practically deployed in typical web development scenarios, we focus on dynamic techniques.

Huang et. al developed WebSSARI, a hybrid approach to securing web applications[36]. The WebSSARI tool uses a static analysis based on type-based information flow to identify possible vulnerabilities in PHP web applications. Their type-based approach operates at a coarse-grain: any data derived from tainted input is considered fully tainted. WebSSARI can insert calls to sanitization routines that filter potentially dangerous content from tainted values before they are passed to security-critical functions. Because we propose techniques for tracking taintedness at a much finer granularity, our system can be more automated than WebSSARI: all we require is that the server uses our modified interpreter PHP to protect all web applications running on the server.

4. Automatic Web Hardening

Our design is based on maintaining precise information about what data is tainted through the processing of a request, and checking that user input sent to an external command or output to a web page contains only safe content. Our automated solution prevents a large class of common security vulnerabilities without any direct effort required from web developers.

The only change from the standard web architecture in Figure 1 is that we replace the standard PHP interpreter with a modified interpreter that identifies which data comes from untrusted sources and precisely tracks how that data propagates through PHP code interpretation (Section a), checks that parameters to commands do not contain dangerous content derived from user input (Section b), and ensures that generated web pages do not contain scripting code created from untrusted input (Section 0).

a. Keeping track of precise taint information

We mark an input from untrusted sources including data provided by client requests as tainted. We modified the PHP interpreter's implementation of the string datatype to include tainting information for string values at the granularity of individual characters. We then propagate taint information across function calls, assignments and composition at the granularity of a single character, hence *precise tainting*. The application of precise tainting enables the prevention of injection attacks and the ability to easily filter output for XSS attacks. If a function uses a tainted variable in a dangerous way, we can reject the call to the function (as is done with SQL queries or PHP system functions) or sanitize the variable values (as is done for preventing cross-site scripting attacks).

Web application developers often remember to sanitize inputs from GET and POSTs, but will omit to check other variables that can be manipulated by clients. Our approach ensures that *all* such external variables, e.g. hidden form variables, cookies and HTTP header information, are marked as tainted. We also keep track of taint information for session variables and database results.

i. Taint strings

For each PHP string, we track tainting information for individual characters. Consider the following code fragment where part of the string $x comes from a web form and the other from a cookie:

```
$x = "Hello " . $_GET['name1'] . ". I am " . $_COOKIE['name2'];
```

The values of $_GET['name1'] and $_COOKIE['name2'] are fully tainted (we assume they are Alice and Bob). After the concatenation, the values of $x and its taint markings (underlined) are: Hello Alice. I am Bob.

ii. Functions

We keep track of taint information across function calls, in particular functions that manipulate and return strings. The general algorithm is to mark strings returned from function as tainted if any of the input arguments are tainted. Whenever feasible, we exploit the semantics of functions and keep track of taintedness precisely. For example, consider the substring function in which taint markings for the result of the substr call depend on the part of the string they select: substr("precise taint me", 2, 10); // ecise tai

iii. Database values and session variables

Databases provide another potential venue for attackers to insert malicious values. We treat strings that are returned from database queries as untrusted and mark them as tainted. While this approach may appear overly restrictive, in the sense that legitimate uses may be prevented, we show in Section 4.3 how precise tainting and our approach to checking for cross-site scripting mitigates this potential problem. Further, if the database is compromised by some other means, the attacker is still unable to use the compromised database to construct a cross-site scripting attack.

The stateless nature of HTTP requires developers to keep track of application state across client requests. However, exposing session variables to clients would allow attackers to manipulate applications. Well-designed web applications keep session variables on the server only and use a session id to communicate with clients. We modified PHP to store taint information with session variables.

b. Preventing command injection

The tainting information is used to determine whether or not calls to security-critical functions are safe. To prevent command injection attacks, we check that the tainted information passed to a command is safe. The actual checking depends on the command, and is designed to be precise enough to prevent all command injection attacks from succeeding while allowing typical web applications to function normally when they are not under attack.

i. PHP injection

To prevent PHP injection attacks we disallow calls to potentially dangerous functions if any one of their arguments is tainted. The list of functions checked is similar to those disallowed by Perl and Ruby's taint mode[37,38] and consists of functions that treat input strings as PHP code or manipulate the system state such as system calls, I/O functions, and calls that are directly evaluated.

ii. SQL injection

Preventing SQL injections requires taking advantage of precise taint information. Before sending commands to the database, e.g. mysql_query, we run the following algorithm to check for injections:

1. Tokenize the query string; preserve taint markings with tokens.
2. Scan each token for identifiers and operator symbols (ignore literals, i.e., strings, numbers, boolean values).
3. Detect an injection if an operator symbol is marked as tainted. Operator symbols are ,()[].;:+-*/\%^<>=~!?@#&|`
4. Detect an injection if an identifier is tainted and a keyword. Example keywords include UNION, DROP, WHERE, OR, AND.

Using the example from Section 2.a:

```
$cmd="SELECT user FROM users WHERE user = ' " . $user
    . '" AND password = ' " . $password . " ' ";
```

The resulting query string (with $user set to ' OR 1 = 1 ; -- ') is tainted as follows: SELECT user FROM users WHERE user = ' ' OR 1 = 1 ; -- ' AND password = 'x'. We detect an injection since OR is both tainted and a keyword.

iii. Preventing cross-site scripting

Our approach to preventing cross-site scripting relies on checking generated output. Any potentially dangerous content in generated HTML pages must contain only untainted data. We modify the PHP output functions (print, echo, printf and other printing functions) with functions that check for tainted output containing dangerous content. The replacement functions output untainted text normally, but keep track of the state of the output stream as necessary for checking. For a contrived example, consider an application that opens a script and then prints tainted output: print "<script>document.write ($user)</script>";

An attacker can inject JavaScript code by setting the value of $user to a value that closes the parenthesis and executes arbitrary code: " me");alert("yo". Note that the opening script tag could be divided across multiple print commands. Hence, our modified output functions need to keep track of open and partially open tags in the output. We do not need to parse the output HTML completely (and it would be unadvisable to do so, since many web applications generate ungrammatical HTML).

Checking output instead of input avoids many of the common problems with ad hoc filtering approaches. Since we are looking at the generated output any tricks involving separating attacks into multiple input variables or using character encodings can be handled systematically. Our checking involves whitelisting safe content whereas blacklisting attempts to prevent cross-site scripting attacks by identifying known dangerous tags, such as <script> and <object>. The latter

fails to prevent script injection involving other tags. For example, a script can be injected into the apparently harmless <b> (bold) tag using parameters such as onmouseover.

Our defense takes advantage of precise tainting information to identify web page output generated from untrusted sources. Any tainted text that could be dangerous is either removed from the output or altered to prevent it being interpreted (for example, replacing < in unknown tags with <). Our conservative assumptions mean that some safe content may be inadvertently suppressed; however, because of the precise tainting information, this is limited to content that is generated from untrusted sources.

5. Conclusion

We have described a fully automated, end-to-end approach for hardening web applications. By exploiting precise tainting in a way that takes advantage of program language semantics and performing context-dependent checking, we are able to prevent a large class of web application exploits without requiring any effort from the web developer. Initial measurements indicate that the performance overhead incurred by using our modified intepreter is less than 10%.

Effective solutions for protecting web applications need to balance the need for precision with the limited time and effort most web developers will spend on security. Fully automated solutions, such as the one described in this paper, provide an important point in this design space.

6. References

1. N. Weidenfeld, *Security Hole Found in Gmail*, (27 October 2004); http://net.nana.co.il/Article/?ArticleID=155025&sid=10.
2. *Report of Ebay Cross-Site Scripting Attack*; http://securityfocus.com/archive/82/246275.
3. *Remotely Exploitable Cross-Site Scripting in Hotmail and Yahoo*, (March 2004); http://www.greymagic.com/security/advisories/gm005-mc/.
4. EyeonSecurity, Microsoft Passport Account Hijack Attack: Hacking Hotmail and More, *Hacker's Digest*.
5. Y.-W. Huang *et al.*, Web Application Security Assessment by Fault Injection and Behavior Monitoring, *Proc. of the World Wide Web Conference (WWW 2003)*, (May 2003).
6. M. Benedikt *et al.*, Veriweb: Automatically Testing Dynamic Web Sites, *Proc. of the World Wide Web Conference*, (May 2002).
7. F. Ricca, and P. Tonella, Analysis and Testing of Web Applications, *Proc. of the IEEE International Conference on Software Engineering*, (May 2001).
8. J. Valade, *Php & Mysql for Dummies*, (Wiley Publishing, 2002).
9. *Netcraft Survey*, (January 2005); http://news.netcraft.com/.
10. JeiAr, *Phpgedview Php Injection*, (Jan 2004); http://xforce.iss.net/xforce/xfdb/14205.
11. Gentoo, *Gallery Php Injection*, (February 2004); http://www.linuxsecurity.com/advisories/gentoo_advisory-4015.html.

12. K. Więsek, Gonicus System Administrator Php Injection, (February 2003).
13. *Santy Worm Used Google to Spread*, (23 December 2004); http://newsfromrussia.com/world/2004/12/23/57537.html.
14. N. Symbolon, *Phpmyadmin Critical Bug*; http://xforce.iss.net/xforce/xfdb/16542.
15. D. Litchfield, *Sql Server Security*, (McGraw-Hill Osborne Media, 2003).
16. K. Spett, "Sql Injection: Are Your Web Applications Vulnerable?" (SPI Labs White Paper, 2002).
17. L. Armstrong, *Phpnuke Sql Injection*, (20 February 2003).
18. *Improved Mysql Extensions*; http://www.php.net/manual/en/ref.mysqli.php.
19. *Malicious Html Tags Embedded in Client Web Requests*, (February 2, 2000); http://www.cert.org/advisories/CA-2000-02.html.
20. G. Hoglund, and G. McGraw, *Exploiting Software: How to Break Code*, (Addison-Wesley, 2004).
21. R. Ivgi, *Cross-Site-Scripting Vulnerability in Microsoft.Com*, (4 October 2004).
22. J. Ley, *Simple Google Cross Site Scripting Exploit*, (17 October 2004).
23. D. Scott, and R. Sharp, Abstraction Application-Level Web Security, *Proc. of the WWW*, (May 2002).
24. *Interdo Web Application Firewall*; http://www.kavado.com/products/interdo.asp.
25. Teros, Inc., *Teros-100 Application Protection System*, (2004); http://www.teros.com/products/aps100/aps.shtml.
26. T. Dyck, *Review: Appshield and Review: Teros-100 Aps 2.1.1*, (May 2003); http://www.eweek.com/article2/0,3959,1110435,00.asp.
27. Tenable Network Security, *Nessus Open Source Vulnerability Scanner Project*, (2005); http://www.nessus.org.
28. J. Offutt *et al.*, Bypass Testing of Web Applications., *Proc. of the IEEE International Symposium on Software Reliability Engineering*, (November 2004).
29. G. S. Kc *et al.*, Countering Code-Injection Attacks with Instruction-Set Randomization., *Proc. of the ACM Computer and Communication Security (CCS)*, (October 2003).
30. S. W. Boyd, and A. D. Keromytis, Sqlrand: Preventing Sql Injection Attacks, *Proc. of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, (June 2004).
31. A. Sabelfeld, and A. C. Myers, Language-Based Information-Flow Security, *IEEE Journal on Selected Areas in Communications* (January 2003).
32. D. E. Bell, and L. J. LaPadula, *Secure Computer Systems: Mathematical Foundations* Mtr-2547, (MITRE Corporation, 1973).
33. K. J. Biba, *Integrity Considerations for Secure Computer Systems* Esd-Tr-76-372, (USAF Electronic Systems Division, 1977).
34. U. Shankar *et al.*, Detecting Format-String Vulnerabilities with Type Qualifiers, *Proc. of the USENIX Security Symposium*.
35. D. Evans, and D. Larochelle, Improving Security Using Extensible Lightweight Static Analysis, *IEEE Software* (January/February 2002).
36. Y.-W. Huang *et al.*, Securing Web Application Code by Static Analysis and Runtime Protection, *Proc. of the World Wide Web Conference*, (May 2004).
37. *Perl 5.6 Documentation: Perl Security*; http://www.perldoc.com/perl5.6/pod/perlsec.html.
38. D. Thomas *et al.*, *Programming Ruby: The Pragmatic Programmer's Guide*, (Pragmatic Programmers, ed. Second, 2004).

Appendix G: Derandomizing Attacks

Attack on Instruction Set Randomization diversity technique

Where's the FEEB?
The Effectiveness of Instruction Set Randomization

Ana Nora Sovarel David Evans Nathanael Paul
University of Virginia, Department of Computer Science

http://www.cs.virginia.edu/feeb

Abstract

Instruction Set Randomization (ISR) has been proposed as a promising defense against code injection attacks. It defuses all standard code injection attacks since the attacker does not know the instruction set of the target machine. A motivated attacker, however, may be able to circumvent ISR by determining the randomization key. In this paper, we investigate the possibility of a remote attacker successfully ascertaining an ISR key using an incremental attack. We introduce a strategy for attacking ISR-protected servers, develop and analyze two attack variations, and present a technique for packaging a worm with a miniature virtual machine that reduces the number of key bytes an attacker must acquire to 100. Our attacks can break enough key bytes to infect an ISR-protected server in about six minutes. Our results provide insights into properties necessary for ISR implementations to be secure.

1. Introduction

In a code injection attack, an attacker exploits a software vulnerability (often a buffer overflow vulnerability) to inject malicious code into a running program. Since the attacker is able to run arbitrary code on the victim's machine, this is a serious attack which grants the attacker all the privileges of the compromised process.

In order for the injected code to have the intended effect, the attacker must know the instruction set of the target. Hence, a general technique for defusing code injection attacks is to obscure the instruction set from the attacker. Instruction Set Randomization (ISR) is a technique for accomplishing this by randomly altering the instructions used by a host machine, application, or execution. By changing the instruction set, ISR defuses all code injection attacks. ISR does not prevent all control flow hijacking attacks, though; for example, the return-to-libc attack [18] does not depend on knowing the instruction set. Much work has been done on the general problem of mitigating code injection attacks, and ISR is one of many proposed approaches. Previous papers have discussed the advantages and disadvantages of ISR relative to other defense strategies [3, 12, 4]. In this paper, we focus on evaluating ISR's effectiveness in protecting a network of vulnerable servers from a motivated attacker and consider properties necessary for an ISR implementation to be secure.

Several implementations of ISR have been proposed. Kc et al.'s design emphasized the possibility of an efficient hardware implementation [12]. They considered a processor in which a special register stores the encryption key. When an instruction is loaded into the processor, it is decrypted by XORing it with the value in the key register.

The processor provides a special privileged instruction for writing into the key register and a different encryption key is associated with each process. The code section of target executable is encrypted with a random key, which is stored in the executable header information so it can be loaded into the key register before executing the program. Kc et al. evaluated their design using the Bochs emulator simulating an x86 processor with a 32-bit key register.

Barrantes et al.'s design, RISE, is not constrained by the need for an efficient hardware implementation [3]. Instead of using an encryption key register, they use a key that can be as long as the program text and encrypt each byte in the code text by XORing it with the corresponding key byte. Encryption is done at load time with a generated pseudo-random key, so each process will have its own, arbitrarily long key. Their implementation used an emulator built on Valgrind [16] to decrypt instruction bytes with the corresponding key bytes when they are executed.

Existing code injection attacks assume the standard instruction set so they will fail against an ISR-protected server. This paper presents a strategy a motivated attacker who is aware of the defense may be able to use to circumvent ISR by determining the key. Our attack is inspired by Shacham et al.'s attack on memory address space randomization [17]. Like ISR, memory address space randomization attempts to defuse a class of attacks by breaking properties of the target program on which the attacker relies (in this case, the location of data structures and code fragments in memory). Shacham et al. demonstrated that the 16-bit key space used by PaX Address Space Layout Randomization [15] could be quickly compromised by a guessing attack.

Many of the necessary conditions for our attack are similar to the conditions needed for Shacham et al.s memory randomization attack. However, since the key space used in ISR defenses is too large for a brute force search, we need an attack that can break the key incrementally. Kc et al. mention the possibility that an attacker might be able to guess parts of the key independently based on the fact that some useful instructions in x86 architecture have only one or two bytes . Our attacks exploit this opportunity.

The key contributions of this paper are:

1. The first quantitative evaluation of the effective security provided by ISR defenses against a motivated adversary.
2. An identification of an avenue of attack available to a remote attacker attempting to determine the encryption key used on an ISR-protected server.
3. Design and implementation of a micro-virtual machine that can be used to infect an ISR-protected server using a small number of acquired key bytes.
4. An evaluation of the effectiveness of two types of attack on a prototype ISR implementation.
5. Insights into the properties necessary for an ISR implementation to be secure against remote attacks.

Next, we describe our incremental key guessing approach. Section 3 provides details on our attack and analyzes its efficiency. Section 4 describes how an attacker could use our attack to deploy a worm on a network of ISR-protected servers. Section 5 discusses the impact of our results for ISR system designers.

2. Approach

The most difficult task in guessing a key incrementally is to be able to notice a good partial guess. Suppose we correctly guess the first two bytes of a four byte key. We would not be able to determine whether or not the guess is correct if the random instruction in the next two bytes executes and causes the program to crash. The result would

be indistinguishable from an incorrect guess of the first two bytes. Even if the next random instruction is harmless, there is a high probability that a subsequently executed instruction will cause the program to crash in a way that is indistinguishable from an incorrect guess.

Our approach to distinguish correct and incorrect partial guesses is to use control instructions. We attempt to inject a particular control instruction with all possible randomization keys. When the guess is correct the execution flow changes in a way that is remotely observable. For an incremental attack to work, the attacker must be able to reliably determine if a partial guess is correct.

For each attempt, there are four possible outcomes:

	Apparently Correct Behavior	Apparently Incorrect Behavior
Correct Guess	Success	*False Negative*
Incorrect Guess	False Positive	*Progress*

Ideally, a correct guess would always lead to distinguishably "correct" behavior, and an incorrect guess would always lead to distinguishably "incorrect" behavior. Given sufficient knowledge of the target system, we should be able to construct attacks where a correct guess never produces an apparently incorrect execution (barring exogenous events that would also make normal requests fail). However, it is not possible to design an attack with perfect recognition: some incorrect guesses will produce behavior that is remotely indistinguishable from that produced by a correct guess. For example, an incorrect guess may decrypt to a harmless instruction, and some subsequently executed instruction may produce the apparently correct execution behavior.

We present attacks based on two different control instructions: return, a one-byte instruction, and jump, a two-byte instruction. For both attacks, if the guess is incorrect, there is a high probability that executing random instructions will cause the process to crash. If the guess is correct, the attacker will observe different server behavior: recognizable output for the return attack and an infinite loop for the jump attack.

Next we describe conditions necessary for the attacks to succeed, explain how each attack is done, and how an incremental attack can be carried out on a large key. For both attacks, there are situations where an incorrect guess produces the same behavior as a correct guess and complications that arise in guessing larger keys. In Section 3, we discuss those issues in more detail and analyze the expected number of attempts required for each attack.

2.1 Requirements

In order for the attack to be possible, the attacker must have some way of injecting code into the target system. We assume the application is vulnerable to a simple stack-smashing buffer overflow attack, although our attack does not depend on how code is injected into the randomized program. It depends only on a vulnerability that can be exploited to inject and execute code in the running process.

Our attack is only feasible for vulnerabilities where the attacker can inject code to a fixed memory location. In a normal stack-smashing attack, the attacker sometimes cannot determine the exact location where code will be inserted because of variations in system libraries, operating system patches and configurations [13]. A common solution is to pad the injected code with nop instructions, often referred to as a *nop sled* [2]. The attack will succeed as long as the return address is overwritten with an address that is in the range of injected nop instructions. When

building an attack against an application protected by ISR, the attacker cannot use this approach because the encryption masks for the positions where nop instructions should be placed are unknown. Another technique, called a register spring [7], overwrites the return address with the address of an instruction found in the application or a library that will indirectly transfer control to the buffer, such as jmp esp or call eax. These instructions are not likely to appear normally in the code, but it is sufficient for an attacker to locate one of the instructions as operand bytes or overlapping bytes in the code segment. Sapphire used a register spring technique by jumping to a jmp esp found in sqlsort.dll [10].

The 32-bit or longer key typically used for ISR is too large for a practical brute force attack, so we must determine the key incrementally. The attacker must be able to acquire enough key bytes to inject the malicious code before the target program is re-randomized with a different key. Since our attack will necessarily crash processes on the target system, it requires either that application executions use the same randomization key each time the target application is restarted, or that the target application uses the same key for many processes it forks. A typical application that exhibits this property is a server that forks a process to serve each client's request. Since failed guess attempts will usually cause the executing process to crash, the attacker must have an opportunity to send many requests to a server encrypted with the same key. Many servers create separate processes to handle simultaneous requests. For example, Apache (since version 2.0), provides configuration options to allow both multiple processes and multiple threads within each process to handle simultaneous requests [1].

Since our attack depends on being able to determine the correct key mask from observing correct guesses, the method used to encrypt instructions must have the property that once a ciphertext-plaintext pair is learned it is possible to determine the key. The XOR encryption technique used by RISE [3] trivially satisfies this property. Kc et al. suggest two possible randomization techniques: one uses XOR encryption and the other uses a secret 32-bit transposition [12]. The XOR cipher, which is what their prototype implements, is vulnerable to our attack. Our attack would not work without significant modification on the 32-bit transposition cipher. Learning one ciphertext-plaintext pair would reduce the keyspace considerably, but is not enough to determine the transposition. Thus, several known plaintext-ciphertext pairs would be needed to learn the transposition key.

The final requirement stems from the remote attacker's need to observe enough server behavior to distinguish between correct and incorrect guesses. If the attack program communicates with the server using a TCP connection it can learn when the process handling the request crashes because the TCP socket is closed. If the key guess is incorrect, the server process will (usually) crash and the operating system will close the socket. Hence, the server must have a vulnerability along an execution path where normal execution keeps a socket open so the remote attacker can distinguish between the two behaviors. If the normal execution flow would close the connection with the client before returning from the vulnerable procedure, the attacker is not able to easily observe the effects of the injected code. The return attack has additional requirements, described in the next section. In cases where those requirements are not satisfied, the (slower) jump attack can be used.

2.2 Return Attack

The return attack uses the near return (0xc3) control instruction [11]. This is a one byte instruction, so it can be found with at most 256 guesses.

Figure 1 shows the stack layout before and after the attack. The attack string preserves the base pointer, replaces the original return address with the target address where the injected code is located, and places the original return address just below the overwritten address. When the routine returns it restores the base pointer register from the stack and jumps to the overwritten return address, which is now the injected instruction. If the guess is correct, the

derandomized injected code is the return instruction. When it executes, the saved return address is popped from the stack and the execution continues as if the called routine returned normally.

There is one important problem, however. When the guess is correct, the return statement that is executed pops an extra element from the stack. In Figure 1, the star marks the position of the top of the stack in normal case (left) and after the injected code is executed successfully (right). After returning from the vulnerable routine, the stack is compromised because the top of the stack is now one element below where it should be. This means the server is likely to crash soon even after a correct guess since all the values restored from the stack will be read from the wrong location.

Thus, the return attack can only be used to exploit a vulnerability at a location where code that sends a response to the client will execute before the compromised stack causes the program to crash. Otherwise, the attacker will not be able to distinguish between correct and incorrect guesses since both result in server crashes. An obvious problem is caused by a subsequent return. At the next return instruction, corresponding to the return from the method that called the vulnerable method, the actual return address is one element up the stack from the location that will be used. It is very likely that the element on the stack interpreted as the return address will be an illegal memory reference. Even when the memory reference is legal, it is unlikely to jump to a location that corresponds to the beginning of a valid instruction.

So, the return attack can only be used effectively for vulnerabilities in which observable server activity (such as a message back to the attack client) occurs between the guessed return and the first instruction that would cause the server to crash (which at the latest, occurs at the end of the called vulnerable routine, but often occurs earlier). We

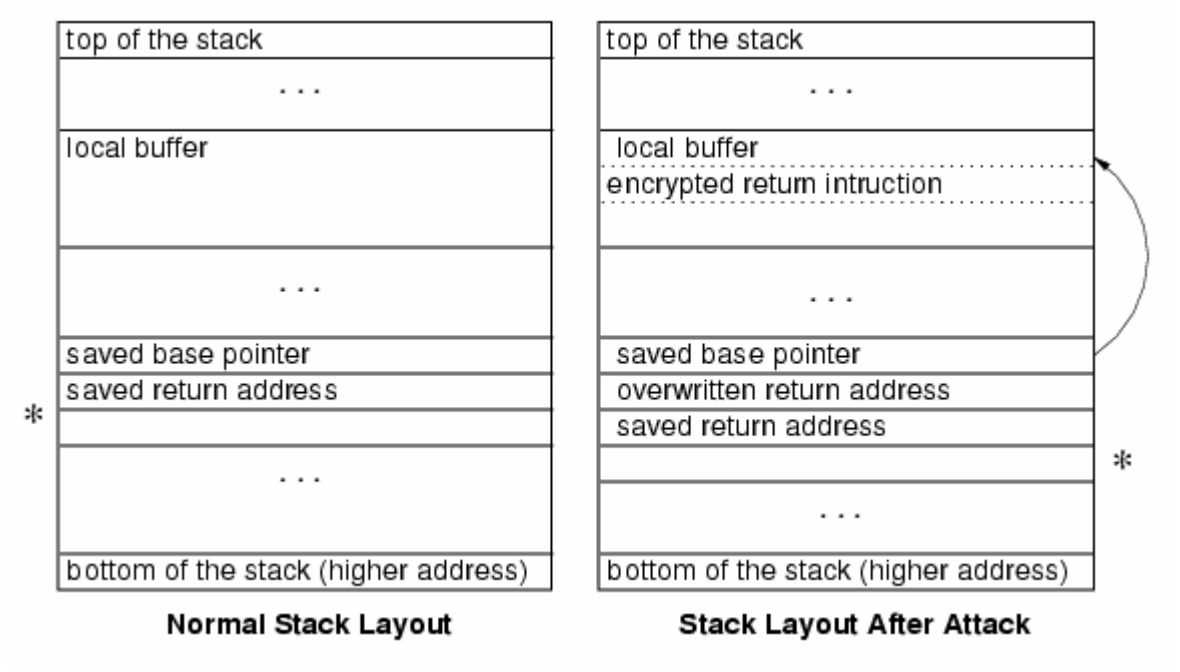


Figure 1. Return attack.

suspect situations where the return attack can be used are rare, but an attacker who is fortunate enough to find such a vulnerability can use it to break an ISR key very quickly.

2.3 Jump Attack

For vulnerabilities where the return attack cannot succeed, we can use the jump attack instead. The advantage of the jump attack is it can be used on any vulnerability where normal behavior keeps a socket open to the client. However, it requires guessing a two-byte instruction, instead of the one-byte return instruction. Another disadvantage of the jump attack is that it produces infinite loops on the server. This slows down server processing for further attack attempts (and may also be noticed by a system administrator). We will present techniques for substantially reducing both the number of guess attempts required and the number of infinite loops created in Section 3.2.

The jump attack is depicted in Figure 2. As with the return attack, the jump attack overwrites the return address with an address on the stack where a jump instruction encrypted with the current guess is placed. The injected instruction is a near jump (0xeb) instruction with an offset -2 (0xfe). If the guess is correct it will jump back to itself, creating an infinite loop. The attacker will see the socket open but receive no response. After a timeout has expired, the attacker assumes the server is in an infinite loop. Usually, an incorrect guess will cause the process handling the request to crash. This is detected by the attacker because the socket is closed before the timeout expires.

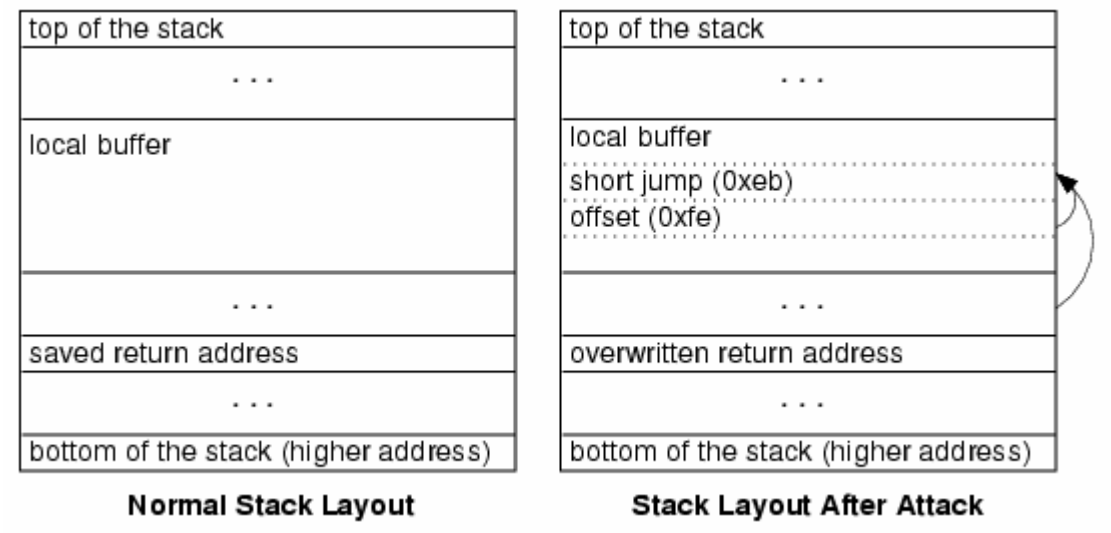


Figure 2. Jump attack.

2.4 Incremental Key Breaking

After the first successful guess, the attacker has obtained the encryption key for one (return attack) or two (jump attack) memory locations. Since other locations are encrypted with different key bits, however, finding one or two key bytes is not enough to inject effective malicious code.

The next step is to change the position of the guessed key byte. For the return attack, we just advance to the next position and repeat the attack using the next position as the return address. With the jump attack, the attacker needs up to obtain the first two key bytes at once, but can proceed in one byte at a time thereafter. On the first attack, shown in the left side of Figure 3, the positions *base* and *base+1* of the attack string are occupied by the jump instruction. On the second attack, we attempt to guess the key at location *base*–1. Since we already know the key for location *base*, we can encode the offset value -2 at that location, and can guess the key for the jump opcode with at most 2^8 attempts.

During the incremental phase of the attack, we decrement the return address placed on the stack for each memory location we guess. At some point the last byte of the address will be zero. This address cannot be injected using a buffer overflow exploit, because it will terminate the attack string before the other bytes can be injected. To deal with this case we introduce an extra jump placed in a position where we already know the encryption key and whose address does not contain a null byte. The return address will point to this jump, which will then jump to the position for which we are trying to guess the key.

When a repeated 32-bit randomization key is used (as in [12]), the number of attempts required to acquire the key using the straightforward attacks would be at most 1024 (4×2^8) for the return attack and 66,048 ($2^{16}+2\times2^8$) for the jump attack (extra attempts may be needed to distinguish between correct guesses and false positives, as explained in the next section). For ISR implementations, such as RISE [3], that do not use short repeated keys the attacker may need to obtain many key bytes before the malicious code can be injected. This cannot be done realistically with the approach described here. Section 3 describes techniques that can be used to make incremental key breaking more efficient. Section 4 explains how many key bytes an attacker will need to compromise to inject and propagate an effective worm.

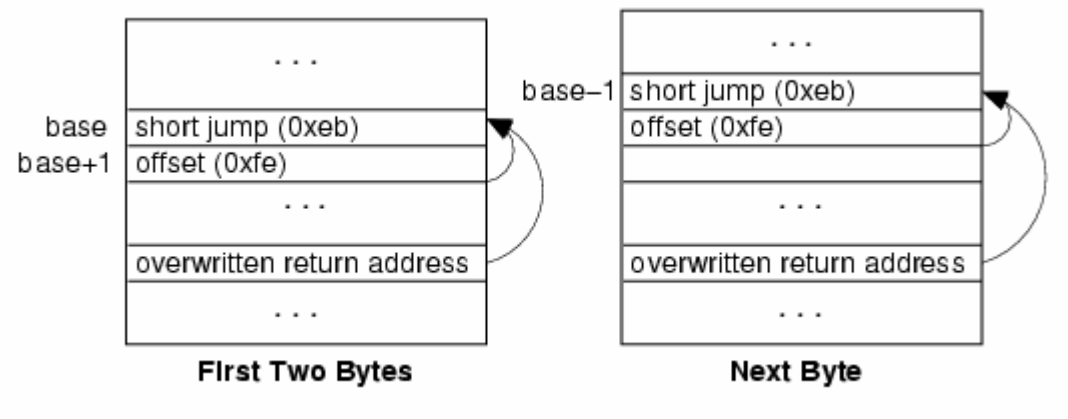


Figure 3. Incremental jump attack.

3. Attack Details and Analysis

The main difficulty in getting the attack to work in practice is that an incorrect guess may have the same behavior as the correct guess. In order to determine the key correctly, the attacker needs to be able to identify the correct key byte from multiple guesses with the same apparently correct server behavior. The next two subsections explain how

false positives can be eliminated with the return and jump attacks respectively. Section 3.3 describes an extended attack that can be used to break large keys.

3.1 Return Attack

There are three possible reasons a return attack guess could produce the apparently correct behavior:

1. The correct key was guessed and the injected instruction decrypted to `0xc3`.
2. An incorrect key was guessed, but the injected instruction decrypted to some other instruction that produced the same observable behavior as a near return.
3. The injected instruction decrypted to an instruction that did not cause the process to crash, and some subsequently executed instruction behaved like a near return.

The first case will happen once in 256 guess attempts.

There are several guesses that could produce the second outcome. The most likely is when the injected instruction decrypts to the 3-byte near return and pop instruction, `0xc2` *imm16*. The near return and pop has the same behavior as the near return instruction, except it will also pop the value of its operand bytes off the stack. Hence, if the current stack height is less than the decrypted value of the the next two bytes on the stack, the observed behavior after a `0xc2` instruction may be indistinguishable from the intended `0xc3` instruction. In the worst case, the stack is high enough for all values to be valid and we will have a false positive corresponding to `0xc2` once every 256 guess attempts.

There are two other types of instructions that can also produce the apparently correct behavior: calls and jumps. In order to produce the near return behavior, the 4-byte offset of the call or jump instruction must jump to the return address. The probability of encountering such a false positive is extremely remote (approximately 2^{-36}). Thus, we ignore this case in our analysis and implementation; this has not caused problems in our experiments.

Given that we observe the return behavior, we can estimate the probability that the correct mask was guessed. We use p_h to represent the probability an arbitrarily long random sequence of bits will start with a *harmless* instruction. We consider any instruction that does not cause the execution to crash immediately after executing it to be harmless (even though it may alter the machine state in ways that cause subsequent instruction to produce a crash). Instruction lengths vary, so determining whether a given injected byte is harmless may depend on the subsequent bytes on the stack. The value of p_h depends on the current state of the execution. Whether or not a given instruction produces a crash depends on the execution's address space, as well as the current values in registers and memory.

We use p_r to represent the probability a random sequence of bits on the stack exhibits the same behavior as the near return instruction, thus capturing cases 1 and 2 above. As we have defined it, the harmless instructions include instructions that behave like the near return. We use $p_{hnr} = p_h - p_r$ to denote the probability random bits correspond to a harmless instruction that does not behave like a near return. Then, we can estimate the probability that a guess produces the apparently correct behavior as:

$$p_{returns} = p_r \sum_{k=0}^{\infty} p_{hnr}^k = \frac{p_r}{(1-p_{hnr})}$$

Given that we observe the correct behavior for some guess, the conditional probability that the guess was actually correct is:

$$\frac{p_{correct}}{p_{returns}} = \frac{(1-p_{hnr})}{(256 * p_r)}$$

The actual values of p_h and p_r depend on the execution state. For our test server application (described in Section 5.1), we compute p_r as 1/256 (probability of guessing 0xc3) + 1/256 (probability of guessing 0xc2) $\times$ $10588/2^{16}$ (fraction of immediate values that do not cause a crash) = 0.00454. In our experiments (described in Section 5.3), we observed the apparently correct behavior with probability 0.0073. The false positive probability is 0.0034. From this, we estimate p_h = 0.43. Thus, 57% of the time an execution will crash on the first random instruction inserted.

Eliminating False Positives

For each memory location for which we want to learn the randomization key, a straightforward implementation guesses all 255 possibilities. We cannot guess the mask 0xc3 using a string buffer overflow attack, since this would require inserting a null byte. If none of the 255 attempts produce the return behavior, we conclude that the actual mask is 0xc3.

If more than one guess produces the apparently correct behavior, we place a known harmless instruction at the guessed position followed by a previously injected guess that produced the return behavior at the next stack position as shown in Figure 4. If this attempt does not exhibit the apparently correct behavior, we can safely eliminate the guessed mask since we know the injected byte did not decrypt to a harmless one-byte instruction as expected. Note that we do not need to know the exact mask for the next position, just a guess we have previously learned produces the return behavior at that location. This approach allows us to distinguish correct guesses from false positives at all locations except for the bottom address (the first one we guess since we are guessing in reverse order on the stack). In cases where multiple guesses are possible for the bottom location, we use its guessed mask only to eliminate false positives in the other guesses, but do not use that location to inject code.

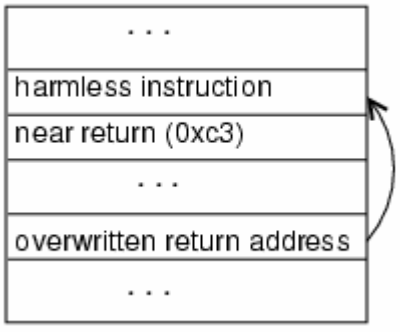


Figure 4. Eliminating false positives.

Harmless instructions help us eliminate false positives for two reasons. If the guess is correct they have known behavior; otherwise, they may decrypt to either a harmful instruction or to an instruction with a different size that will alter the subsequent instructions. In the second case, it is possible to still produce the apparently correct behavior when the mask guess is incorrect. Hence, we learn conclusively when a mask is incorrect, but still cannot be sure the guess is correct just because it exhibits the correct behavior.

The number of useful harmless one-byte instructions is limited by the density of x86 instruction set. If there are groups of harmless instructions with similar opcodes, it is hard to differentiate between them. Harmless instructions are only useful if an incorrect mask guess encrypts the guessed harmless instruction to an instruction that causes a crash. For example, if we use as harmless instructions a group of similar instructions such as clear carry flag (0xf8), clear direction (0xfc), complement carry flag (0xf5), set carry flag (0xf9), set direction flag (0xfd), the number of masks eliminated is in most of the cases is the same as if we had use only one of these instructions. Our attack uses three disparate one-byte harmless instructions: nop (0x90), clear direction (0xfc), and increment ecx register (0x42).

For a given set of possible masks it would be possible to determine a minimal set of distinguishing harmless instructions, however this would add substantially to the length and complexity of the attack code. Instead, in the rare situations where the three selected one-byte harmless instructions are unable to eliminate all but one of the guessed masks, we use harmless two-byte instructions, of which there are many. This approach works for all locations except the next-to-bottom address. In the rare situations when it is not possible to determine the correct mask for this location, we can simply start the injected attack code further up the stack.

Using harmless one-byte and two-byte instructions we are able to reduce the number of apparently correct masks to at most two. We cannot handle the case where the first instruction decrypts to a near return and pop instruction (0xc2 *imm16*) using this elimination process described because the near return (0xc3) and near return and pop (0xc2) opcodes differ by only their final bit. There is no harmless x86 instruction we can use to reliably distinguish them. When a harmless instruction is encrypted with an incorrect mask and decrypted with the correct masks, the opcode of the instruction executed differs only by one bit from the guessed harmless instruction. It is likely that this instruction will be a harmless instruction too.

To distinguish between the two forms of near return we place the bytes 0xc2 0xff 0xff on the stack using the guessed masks. This is a near return which pops 65,535 bytes from the stack. For many target vulnerabilities (including our test server), this is enough to generate a crash. To use this approach, we need to already know the next two masks on the stack. This is not a problem because we start elimination from the bottom of the stack. The first two times we apply elimination with 0xc2 we have to execute an attempt for each combination of possible masks of the next two positions. After that, we know the correct masks for the locations where we place the 0xffff.

For target applications for which popping 65,535 bytes from the stack does not cause a crash, we can use another type of elimination. After we guess enough bytes, we use a jump instruction to eliminate incorrect masks. We place a jump instruction with its offset encrypted using one of the apparently correct guessed masks. The jump instruction when the mask is correct will cause a jump to a memory location where a near return is placed.

Once we have determined six or more masks, we can take advantage of additional injected instructions to further minimize the likelihood of false positives and improve guessing efficiency. These techniques are similar for both the return and jump attacks, and are described in Section 3.3.

3.2 Jump Attack

Because it involves guessing a 2-byte key and the distinguishing behavior is less particular, the jump attack is more prone to false positives than the return attack. Fortunately, the structure of the x86 instruction set can be used to take advantage of the false positives to improve the key search efficiency.

There are four possible reasons the apparently correct behavior is observed for a jump attack guess:

1. The correct key was guessed and the injected instruction decrypted to a jump with offset -2.
2. The injected guess decrypted to some other instruction which produces an infinite loop.
3. The injected instruction decrypted to a harmless instruction, and some subsequently executed instruction produces an infinite loop.
4. The injected guess caused the server to crash, but because of network lag or server load, it still took longer than the timeout threshold the attacker uses to identify infinite loops.

We can avoid case 4 by setting the timeout threshold high enough, but this presents a tradeoff between attack speed and likelihood of a false positive. A more sophisticated attack would dynamically adjust the timeout threshold. Since case 4 is likely to occur for many guesses and will not occur repeatedly for the same guess, case 4 is usually distinguishable from the other three cases and the attacker can increase the timeout threshold as necessary.

From a single guess, there is no way to distinguish between case 1 (a correct guess) and cases 2 and 3. However, by using the results from multiple guesses, it is possible to distinguish the correct guesses in nearly all instances.

For the second case, there are two kinds of false positives to consider: (1) the opcode decrypted correctly to `0xeb`, but the offset decrypted to some value other than -2 which produced an infinite loop; or (2) the opcode decrypted to some other control flow instruction that produces an infinite loop.

An example of the first kind of false positive is when the offset decrypts to -4 and the instruction at offset -4 is a harmless two-byte instruction. This is not a big problem, since, as we presented in Section 2.3, except for when we are guessing the first two bytes we are encrypting the offset with a known mask. When it does occur in the first two bytes, the attacker has several possibilities. One is to ignore this last byte and use only the memory locations above it. Another possibility is to launch different versions of the injected attack code, one for each possibly correct mask. Sometimes it would be faster to launch four versions of the attack code, one of which will succeed, than to determine a single correct mask at the bottom location.

The second case, where the opcode is incorrect, is more interesting. The prevalence of these false positives, and the structure of the x86 instruction set, can be used to reduce the number of guesses needed. The other two-byte instructions that could produce infinite loops are the near conditional jumps. Like the unconditional jump instruction, the first byte specifies the opcode and the second one the relative offset. There are sixteen conditional jump instructions with opcodes between `0x70` and `0x7f`. For example, opcode `0x7a` is the `JP` (jump if parity) instruction, and `0x7b` is the `JNP` (jump if not parity) instruction. Regardless of the state of the process, exactly one of those two instructions is guaranteed to jump. Conveniently, all the opcodes between `0x70` and `0x7f` satisfy this

complementary property. Thus, for any machine state, exactly 8 of the instructions with opcodes between 0x70 and 0x7f will jump, producing the infinite loop behavior if the mask for the offset operand is correctly guessed. When we find several masks sharing the same high four bits of the first byte that all produce infinite loops, we can be almost certain that those four bits correspond to 0x7.

We can take further advantage of the instruction set structure by observing that if we try both guesses for the least significant bit in the opcode, we are guaranteed that one of the two guesses will produce the infinite loop behavior if the first four bits of the guess opcode are 0x7. That is, if we guess two complementary conditional jump instructions, one of them will produce the infinite loop behavior; it doesn't matter what the other three bits are, since all of the conditional jump opcodes have the same property.

This observation can be used to substantially reduce the number of attempts needed. Instead of needing up to 256 guesses to try all possible masks for the opcode byte, we only need 32 guesses (0x00, 0x10, 0x20, ..., 0xf0, 0x01, 0x11,...,0xf1) to try both possibilities for the least significant bit with all possible masks for the first four bits. Those 32 guesses always find one of the taken conditional jump instructions. Hence, the maximum number of attempts needed to find the first infinite loop (starting with no known masks) is 2^{13} (2^5 guesses for the opcode $\times$ 2^8 guesses for the offset). When the offset is encrypted with a known mask (that is, after the first two byte masks have been determined), at most 32 attempts are needed to find the first infinite loop. The expected number of guesses to find the first infinite loop is approximately 15.75 since we can find it by either guessing a taken conditional jump instruction or the unconditional jump. (This analytical result is approximate since it depends on the assumption that each conditional jump is taken half the time. Since the actual probability of each conditional jump being taken depends on the execution state, the actual value here will vary slightly.)

After finding the first infinite loop producing guess, we need additional attempts to determine the correct mask. The most likely case ($15/16^{th}$s of the time), is that we guessed a taken conditional jump instruction. If this is the case, we know the first four bits unmask to 0x7, but do not know the second four bits. To find the correct mask, we XOR the guess with 0x7 $\oplus$ 0xe and guess all possible values of the second four bits until an infinite loop is produced. This means we have found the 0xeb opcode and know the mask. Thus, we expect to find the correct mask with 8 guesses. The other $1/16^{th}$ of the time, the first loop-producing guess is the unconditional jump instruction. We expect to find two infinite loops within first four attempts. If we find them, we know we guessed the correct mask; otherwise we continue. We expect on average to use 15.75 guesses to find the first infinite loop and 7.75 guesses to determine the correct mask. Hence, after acquiring the first two key bytes, we expect to acquire each additional key byte using less than 24 guesses on average, while creating two infinite loops on the server.

In rare circumstances, the first infinite loop encountered could be caused by something other than guessing an unconditional or conditional jump instruction. One possibility is the loop instruction. The loop instruction can appear to be an infinite loop since it keeps jumping as long as the value in the ecx register is non-zero. When ecx initially contains a high value the loop instruction can loop enough times to exceed the timeout for recognizing an infinite loop. There are several possible solutions: wait long enough to distinguish between the jump and the loop, find a vulnerability in a place where ecx has a low value (an attacker may be able to control the input in such a way to guarantee this), or to use additional attempts with different instructions to distinguish between the loop and jump opcodes. For simplicity, we used the second option: in our constructed server, the ecx register has a small value before the vulnerability.

The other possibility is the injected code decrypts to a sequence of harmless instructions followed by a loop-producing instruction. This is not as much of a problem as it is with the return attack since the probability of two

random bytes decrypting to a loop-producing instruction is much lower than the probability of a single random byte decrypting to a return instruction. Further, when it does occur, the structure of the conditional jumps in the instruction set makes it easy to eliminate incorrect mask guesses. The probability of encountering an infinite loop by executing random instructions was found by Barrantes, et al. to be only 0.02% [3]. However, since we are not guessing randomly but using structured guesses, the probability of creating infinite loops is somewhat higher. In the first step of the attack we generate all possible combinations for first two bytes. An infinite loop is created by an incorrect guess when first byte decrypts to a harmless one-byte instruction, and the second byte decrypts to a conditional or unconditional jump instruction, and the third byte decrypts to a small negative value. In this case both -2 and -3 will create infinite loops. To avoid false positives and increased load on the server, after we find the first infinite loop, we change the sign bit of the third byte. This changes the value to a positive one. If the loop was created by an incorrect mask, when we verify the mask with conditional jumps and fail to find the expected infinite loops we can conclude the mask guess is incorrect.

3.3 Extended Attack

The techniques described so far are adequate for obtaining a small number of key bytes. For ISR implementations that use a short repeated key, such as [12], obtaining a few key bytes is enough to inject arbitrarily long worm code. For ISR implementations that use a long key, however, an attacker may need to acquire thousands of key byte masks before having enough space to inject the malicious code. Acquiring a large number of key bytes with the jump attack is especially problematic since attempts leave processes running infinite loops running on the server. After acquiring several key bytes this way, the server becomes so sluggish it becomes difficult to distinguish guess attempts that produce crashes from those that produce infinite loops.

Once we have learned a few masks, we can improve the attack efficiency by putting known instructions in these positions. With the jump attack, once we have guessed four bytes using short jumps, we change the guessed instruction to a near jump (0xe9). Near jump is a 5-byte instruction that takes a 4-byte offset as its operand. This is long enough to contain an offset that makes the execution jump back to the original return address. Hence, we no longer need to create infinite loops on the server to recognize a correct guess: we recognize the correct guess when the server behaves normally, instead of crashing.

When the server has the properties required by the return attack, we will encounter false positives for the near jump guessed caused by a relative call (0xe8). Since the opcode differers from the near jump opcode in only one bit, we are not able to reliably distinguish between the two instructions using harmless instructions. Instead, we keep both possible masks under consideration until the next position is guessed, and then identify the correct mask by trying each guess for the offset mask. At worst, we need four times as many attempts because it is possible that there are two positions with two possible masks in the offset bytes. Despite requiring more attempts, this approach is preferable to the short jump guessing since it reduces the load on the server created by infinite loops.

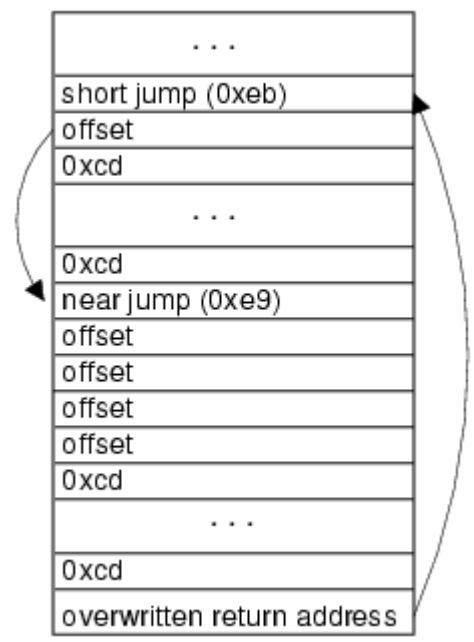


Figure 5. Extended attack.

Once we have acquired eight masks, we switch to the extended attack illustrated in Figure 5. The extended attack requires a maximum of 32 attempts per byte, and expected number of 23.5. The idea is to use a short jump instruction to guess the encryption key for current location with an offset that transfers control to a known mask location where we place a long jump instruction whose target is the original return address. The long jump instruction is a relative jump with a 32 bit offset. Hence, we need to acquire four additional mask bytes before we can use the extended attack with the jump attack.

To eliminate false positives, we inject bytes that correspond to an interrupt instruction in the subsequent already guessed positions. Interrupt is a two-byte instruction (0xcd *imm8*). The second byte is the interrupt vector number. When the guessed instruction decrypts to a harmless instruction, the next instruction executed will be 0xcdcd (INT 0xcd) which causes a program crash. The only value acceptable for the interrupt vector number in user mode when running on a Linux platform is 0x80 [5]. The key is to place enough 0xcd bytes in the region such that when the first instruction decrypts to some harmless non-jump instruction (which could be more than one byte), the next instruction to execute is always an illegal interrupt. Once we have room for six 0xcd bytes, we encounter no false positives.

If any of the masks in this region are 0xcd, we cannot place a 0xcd byte at that location since injecting the necessary instruction which would require injecting a null byte. In this case, we place an opcode corresponding to a two-byte instruction (we use AND, but any instruction would work). The 0xcd will be the second byte of the two-byte instruction. After the two-byte instruction it will find a 0xcd which causes a crash.

The most important advantage of this approach is that the only cases when the server sends the expected response are when (1) the first instruction executed is a taken unconditional jump; or (2) the first instruction executed is a

conditional jump where the condition is true. With the return attack there is a third case: the first instruction executed is a near return. This possibility can be eliminated using the techniques described in Section 3.1.

The other advantage of this attack is that it does not need to create infinite loops on the server. Once we have enough mask bytes to inject a long jump instruction, we can distinguish correct guesses without putting the server in an infinite loop. Instead, the attacker is able to recognize a correct guess when it receives the expected response from the server.

4. Deployment

If the malicious code is small (for example, the Sapphire worm was 376 bytes [9]), we can acquire enough key bytes to inject it directly. This is reasonable if we are attacking a single ISR-protected machine using this approach and can run our attack client code on a machine we control to obtain enough key bytes to inject the malicious code. If the attacker wants to propagate a worm on a network of ISR-protected servers, however, the worm code needs to contain all the code for implementing the incremental key attack also. This may require acquiring more key bytes than can be done without the system administrator noticing the suspicious behavior and re-randomizing the server. Since the ISR-breaking code is inherently complex, even if the malicious payload is small many thousands of key bytes would be needed to inject the worm code.

Our strategy is to instead inject a micro virtual machine (MicroVM) in the region of memory where we know the key masks. The MicroVM executes the worm code by moving small chunks of it at a time into the region where the key masks are known. The next subsections describe the MicroVM and how worm code can be written to work within our MicroVM. In order to make the MicroVM as small as possible we place restrictions and additional burdens on the worm code.

4.1 MicroVM Implementation

The MicroVM is illustrated in Figure 6. At the heart of the MicroVM is a loop that repeatedly reads a block of worm code into a region of memory where the masks are known and executes that code. The code (shown in Appendix A) is 98 bytes long (including the 22 bytes of space reserved for executing worm code).

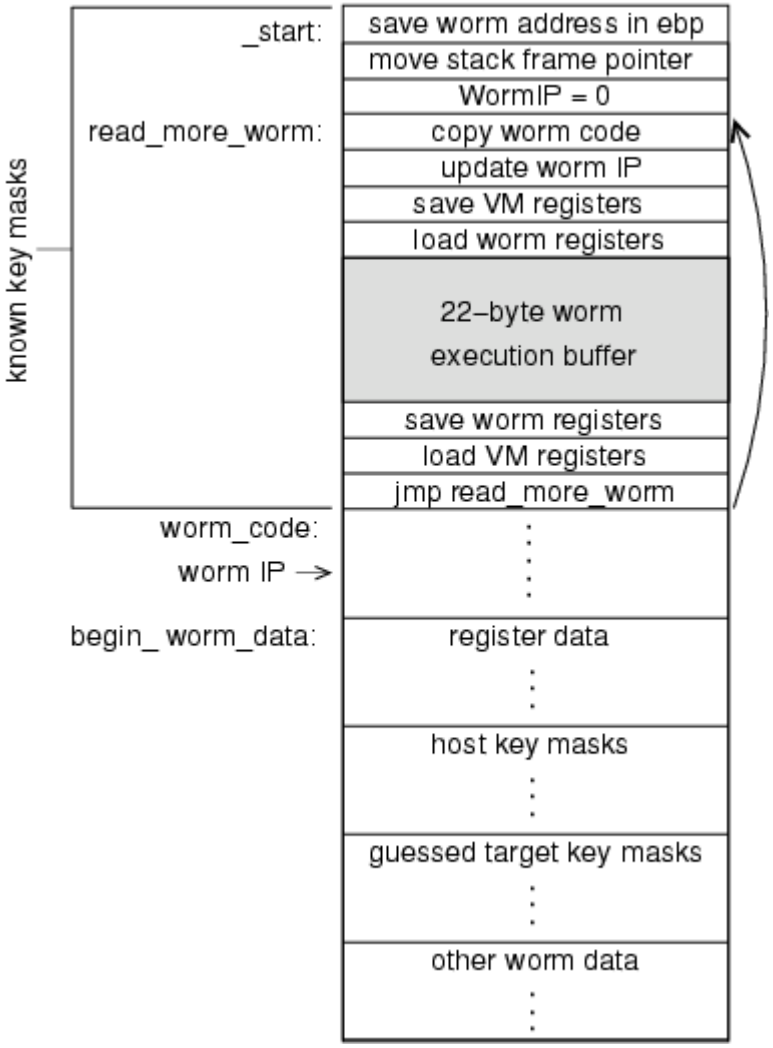


Figure 6. MicroVM.

Before starting the execution loop, the MicroVM initializes the worm instruction pointer (WormIP) to contain 0 to represent the beginning of the worm code. The WormIP stores the next location to read a block of worm code. Next, a block of worm code is fetched by copying the bytes from the worm code (from the WormIP) into an execution buffer inside the MicroVM itself, so that execution can simply continue through the worm code and then back into the MicroVM code without needing a call. The addresses of the beginning of the worm code and worm data space are hardcoded by the worm code into the MicroVM when it is deployed on a new host.

No encryption is necessary when worm code is copied into the execution buffer, since the worm code was already encrypted with known key masks for the worm execution buffer locations where it will be loaded into the worm execution buffer.

Just before the execution of the worm block, the MicroVM pushes its registers on the stack and then restores the worm's registers from the beginning of the worm data region. After the buffer's execution, the MicroVM saves the worm's registers to the worm data region. In the last step, the MicroVM restores its registers and then jumps back to the beginning of the MicroVM code to execute the next block of worm code.

4.2 Worm Code

To work in the MicroVM, the worm code is divided into blocks matching the size of the worm execution buffer (22 bytes in our implementation). No instruction can be split across these blocks, so the worm code is padded with nops as necessary to prevent instructions from crossing block boundaries. The worm code cannot leave data on the execution stack at the end of a block, since the MicroVM registers are pushed on the stack just before the worm execution begins. To use persistent data, the worm must write into locations in the worm data space instead of using the execution stack.

The most cumbersome restrictions involve jumps. Any jump can occur within a single worm block, but jumps that transfer control to locations outside the buffer must be done differently since all worm code must executed at known mask locations in the worm buffer. Our solution is to require that all jumps must be at the end of a worm code block, and all jump targets must be to the beginning of a worm code block. Instead of actually executing a jump, the worm code updates the value of the WormIP (which is now stored in a known location in memory, and will be restored when the MicroVM resumes) to point to the target location, and then continues into the MicroVM code normally so the target block will be the next worm code block to execute. To implement a conditional jump, we use a short conditional jump with the opposite condition within the worm buffer to skip the instruction that updates the WormIP when the condition is unsatisfied.

4.3 Propagation

To propagate, the worm uses the techniques described in Section 3 to acquire enough key bytes to hold the MicroVM. Those key bytes are stored in the worm data region. The MicroVM code is 98 bytes long so at least 98 key bytes are needed. We may need to acquire a few additional key bytes to avoid needing to place null bytes in the attack code. If the mask found for a given location matches the bytes we want to put there, we instead put a nop instruction at that location and obtain an extra key byte. As long as the masks are randomly distributed, two or fewer will be sufficient over 99% of the time, so we can nearly always inject the worm once 100 key bytes have been acquired.

To generate an instance of the worm for a new key, we XOR out the old key bytes from the worm code and XOR in the new key bytes. To support this, the propagated worm data includes th host's acquired mask bytes. As with the injected MicroVM code, we need to worry about the impossibility of injecting null bytes. We insert nops in the injected worm code as necessary to avoid null bytes. If the added nops would cause a worm code block to exceed the available space, we need to create a new block and move the overflow instructions into that block. Jump targets in the worm code may need to be updated to reflect insertion of the new block.

5. Results

To test our attack we built a small echo server with a buffer overflow vulnerability. The application waits for a client to connect. When the client connects, the server forks a process to process its request. The next step is to call a method which has a local buffer that can be overflowed. This method reads the request from the client and writes

back an acknowledgment message. After this method call the application sends a termination message (“Bye”) and closes the socket. Although we use a contrived vulnerability to make the attack easier to execute and analyze, similar vulnerabilities are found in real applications.

5.1 Attack Client

The attack client structure is the same for both the jump and return attacks. For each guess attempt, the attack client (1) opens a socket to the server, (2) builds an attack string, (3) writes it to the socket, (4) reads the acknowledgment, (5) installs an alarm signal handler, (6) sets up an alarm, and (7) reads the termination message or handles the alarm signal. The return attack recognizes a possibly correct guess when it receives the termination message in step 7; the jump attack recognizes a possibly correct guess when the alarm signal handler is called before the socket is closed.

The attack strategy used for different key bytes is depicted in Figure 7. The number of key bytes guessed by the attack is denoted by *size*. For vulnerabilities suitable for the return attack, the first eight positions are guessed using the return instruction. The rest are guessed using the extended short jump attack (expected 23.5 attempts per byte). For the jump attack, the first two key bytes acquired have positions *size-1* and *size-2*. We guess those two bytes simultaneously, using the 2-byte jump instruction to create an infinite loop. The next two bytes are guessed separately using the jump instruction to create an infinite loop. After the fourth byte is acquired, we do not (intentionally) create any more infinite loops. For the next six bytes, we use near jump, with a worst case of 1024 attempts per byte. After this position, we use the extended short jump attack.

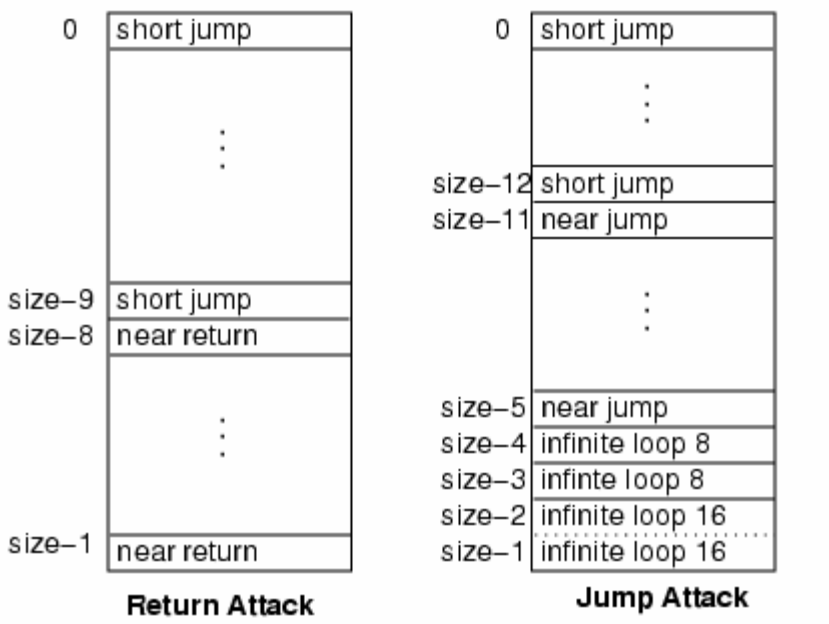


Figure 7. Guessing strategies.

For the attack client to be efficient there are some constraints on the address where the attack starts. For both attacks the address has to be far enough from the next smaller address which has null as its last byte so we have enough space to place two short jump instructions, and a sufficient number of illegal opcodes. As long as the vulnerable buffer is sufficiently large, the attack client can find a good location to begin the attack.

We ran our client normally, not inside the MicroVM. Hence, our results correspond to the time needed to launch the initial attack on the first ISR-protected server. The attack time would increase for later infections because of the additional overhead associated with executing in the MicroVM.

5.2 Target

We executed our attack on our constructed vulnerable server protected by RISE [3]. The RISE implementation presents a major difficulty in executing our attack because of the way it implements fork, pthreads and randomization keys. This necessitated a small modification to RISE in order for our attack to succeed. Other ISR implementations, however, may be vulnerable to our attack without needing this modification.

RISE uses a different key to randomize an application each time it is started. Since the attack causes the server to crash, the attack can only work against a server that forks separate processes to handle client requests. Valgrind [16] (the emulator modified to implement RISE) implements pthreads to use only one process. Thus, if the attack crashes a thread, then the entire server will crash and the next execution will use a different randomization key. So, our attack will only work against a server that forks separate processes.

When RISE loads an application, a cache data structure is initialized that holds the key mask for each instruction address that has been loaded. There is a different randomization key byte for each byte in the text segment, and the mask value is stored in the cache the first time the corresponding instruction address is loaded.

The fork call is forwarded to the operating system and results in a new child process running the emulator. When the injected instructions execute, the child process will determine that no mask has been initialized for the address on the stack and it will generate a new one. Hence, the child process will share the same randomization key for the addresses already loaded in memory at fork time, but for the addresses it accesses later it will use it’s own key. This is problematic since the incremental attack only works if multiple attempts can be launched attacking the same key.

Perhaps an attacker could control the execution enough to ensure that the necessary masks are initialized before the child process forks to ensure they would be the same on all executions. This would only happen, however, if the server legitimately ran code on the stack before reaching the vulnerability. Hence, the RISE implementation of ISR is not vulnerable to our attack.

In order to experiment with our attack, we modified RISE to initialize the masks for all used instruction addresses before the child process forks to ensure that all child processes have the same key. Obviously, a real attacker would not have this opportunity.

In addition to the problems caused by the emulator itself, we encountered others caused by the operating system. The Fedora Linux distribution has address space layout randomization enabled by default. For our experiments, we disabled this defense. Attacks on systems using both address and instruction randomization pose additional challenges that are beyond the scope of this paper.

5.3 Experimental Results

Table 1, Figure 8 and Figure 9 summarize the results from our experiments. The target and client ran on separate Linux dual AMD Athlon XP 2400+ machines. connected to the same network switch. For key lengths up to 128, we executed 100 trials; for longer keys, we executed 20 trials. In all cases, our attacks are nearly always able to obtain the correct key and the attack completes in under one hour, even for acquiring a 4096-byte key using the jump attack. A successful attack is an execution in which the attack client correctly guesses the desired number of key bytes. Every key byte must be correct for us to consider the attack a success.

Key Bytes	*Attempts*	*Attempts per Byte*	*Infinite Loops*	*Success Rate (%)*	*Time (s)*
2	3983	1991.6	3.86	98	138.3
4	4208	1052.1	8.11	99	207.9
32	7240	226.3	8.28	98	283.6
100	8636	86.4	9.15	100	365.6
512	18904	36.9	8.31	95	627.4
1024	30035	29.3	7.90	100	974.3
4096	102389	25.0	8.36	95	2919.4

Table 1. Jump attack results (averages over all trials).

The experiments confirm the analytical predictions regarding the decrease of number of attempts per byte as key length increases. After breaking the first 12 bytes, fewer than 24 guess attempts are required per byte to acquire additional key bytes. On average, we can break a 100-byte key (enough to inject our MicroVM code) in just over six minutes with the jump attack. The return attack is faster, and requires less than two minutes. The difference is the additional approximately 4000 expected attempts the jump attack needs to guess the first two bytes simultaneously. The other difference is the increased time per attempt needed for the jump attack stemming from the infinite loops running on the server. The return attack produces an infinite loop on the server only in the unlucky circumstances when a random instruction happens to produce an infinite loop. In our experiments, the average number of infinite loops created during a return attack is 0.76. Rarely, we may be unlucky and create many infinite loops with the return attack (such as was the case for the extreme maximum time value in breaking a 4-byte key in Figure 8). The jump attack must create several infinite loops to guess the first key bytes. The actual number of loops created, shown in Table 1, varies depending on the number of apparently correct offset values.

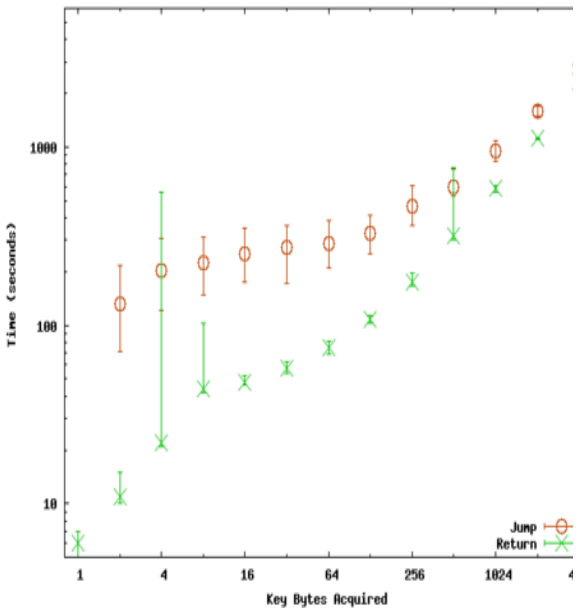


Figure 8. Time to acquire key bytes.

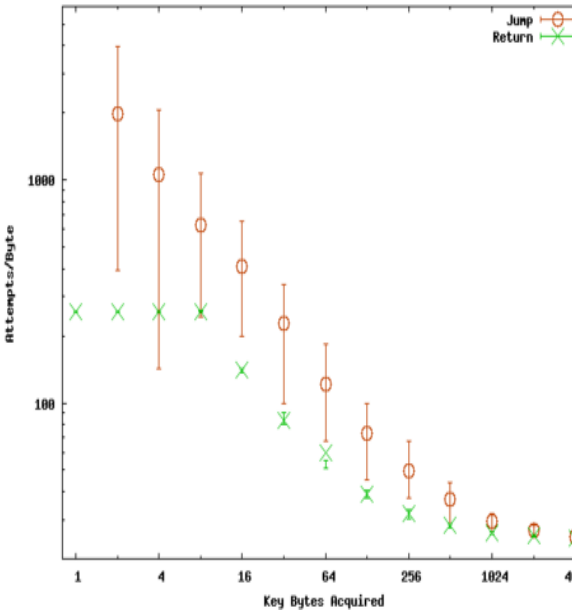


Figure 9. Attempts per byte.

Times are wall-clock times measured by the client for the duration of the attack. The marked points are the median values and the bars show the 95th percentile maximum and minimum results over all trials.

In our initial experiments, we had surprising results where trials guessing 32-byte keys were always taking longer than guessing 2048-byte keys. The bytes placed on the stack during the near jump phase of the 32-byte attack (guessing mask bytes 5 through 11) included an 0xfe byte. This meant if the guessed instruction decrypted to a harmless instruction the execution could fall through to the 0xfe instruction and generate an infinite loop. Instead of the typical number of infinite loops, over 20 infinite loops were being created. This increased the server load enough to make the 32-byte key trials take longer than the 2048-byte keys. We modified the attack client to avoid this problem by making it select an address for starting the guessing that ensures 0xfe will not appear in the near jump offset.

In a few cases, our attack was not able to determine the correct key. The failures are caused by the inability to use certain masks because injecting the desired encrypted byte would require placing a null byte on the stack, which will cause the attack string to end before the return address is overwritten. Workarounds are possible, and necessary for the common cases. For example, in the return attack we will get an incorrect mask when a position has an apparently correct guess, but the mask is the return opcode. We assume 0xc3 is the correct mask when all the other 255 masks fail to produce the return behavior. Similarly, for the jump attack we will have false positives when the mask for the last position guessed is 0xfe. Our experimental results demonstrate that with the strategies we use the likelihood of incorrect guesses is small enough that it is not worth increasing the length and complexity of the attack code to deal with the rare special cases.

6. Discussion

Our attack is essentially a chosen-ciphertext attack on an XOR encryption scheme. If we obtain a known ciphertext-plaintext pair with such a cipher, obtaining the encryption key is a trivial matter of XORing the plaintext and ciphertext. The challenge is obtaining a known plaintext. We do not actually obtain the plaintext for a given ciphertext guess, but instead obtain clues from the remotely observed behavior. After enough guesses, though, we can reliably determine the corresponding plaintext for an input ciphertext, and acquire the key.

This suggests some simple modifications to ISR implementations that can be used to make incremental guessing attacks much less likely to succeed. Our attack strategy would not work against any ISR scheme that uses an encryption algorithm that is not susceptible to a simple known plaintext-ciphertext attack. Any modern block encryption algorithm (such as AES [8]) satisfies this property. Unfortunately, the performance overhead of decrypting executing instructions with such an algorithm may be prohibitive. A more efficient but less secure alternative might be to randomly map each 8-bit value to a value using a lookup table. Combining this with the XOR encryption would make incremental key attacks like we propose much more difficult since it would hide the structure of the actual instruction set from the adversary.

The other property our attack relies on that is easily altered is the need to make many attempts that crash a process against a binary randomized using the same key. RISE is largely invulnerable to our attack because of the way it uses different randomization keys for forked processes. If re-randomizing is inexpensive, an implementation that re-randomizes the binary after every process or thread crash would not be susceptible to incremental key breaking attacks. This approach, however, does make the server increasingly vulnerable to denial-of-service attacks since all an attacker needs to do to force the server to shutdown and restart itself with a new randomization key is to crash a single thread.

The details of our attacks are heavily dependent on the x86 instruction set. In particular, our attacks rely on the presence of short (one or two-byte) control instructions and short harmless instructions, and benefit substantially from the structure of the conditional jump instructions. For any RISC architecture with fixed instruction length, the minimum number of key bits that must be guessed at once is determined by the instruction length. Most RISC architectures use instruction lengths of at least 32 bits, which is probably too long to realistically guess using a brute-force approach.

7. Conclusion

We have demonstrated that servers protected using ISR may be vulnerable to an incremental key-breaking attack. Our attack enables a remote attacker to acquire enough key bytes to inject an arbitrarily long worm in an ISR-protect server in approximately six minutes using the jump attack.

Our results apply only to the use of ISR at the machine instruction set level; our techniques could not be used directly to attack ISR defenses for higher-level languages such as SQL [6] and Perl [12].

Our results indicate that doing ISR in a way that provides a high degree of security against a motivated attacker is more difficult than previously thought. The most efficient ISR proposals, such as the repeated 32-bit XOR key, provide little security under realistic conditions. This does not mean ISR is no longer a promising defense strategy, but it means designers of ISR systems must consider carefully how effectively their randomization thwarts possible strategies for remotely determining the randomization key.

Acknowledgments

The authors thank Gabriela Barrantes for generously providing the RISE implementation for our experiments. We are grateful to Stephanie Forrest, Patrick Graydon, and Trent Jaeger for providing useful and insightful comments on early versions of this paper. This work benefited from fruitful discussions with Lee Badger, Steve Chapin, Jack Davidson, Dragos Halmagi, Xuxian Jiang, Angelos Keromytis, John Knight, David Mazières, Cristina Nita-Rotaru, Anh Nguyen-Tuong, Fred Schneider, Jeffrey Shirley, Mary Lou Soffa, Peter Szor, Dan Williams, Dongyan Xu, and Jinlin Yang. We thank Andrew Barrows, Jessica Greer, Scott Ruffner, and Jing Yang for technical assistance, and the Guadalajara Restaurant for Special Lunch #3. This work was supported in part by grants from the DARPA Self-Regenerative Systems Program (FA8750-04-2-0246) and the National Science Foundation (through grants NSF CAREER CCR-0092945 and NSF ITR EIA-0205327).

References

[1] Apache Software Foundation. *Apache MPM Worker.* Apache HTTP Server Version 2.0 Documentation. http://httpd.apache.org/docs 2.0/mod/worker.html

[2] Murat Balaban. *Buffer Overflows Demystified*. http://www.enderunix.org/documents/eng/bof-eng.txt

[3] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Intrusion detection: Randomized instruction set emulation to disrupt binary code injection attacks. *10th ACM Conference on Computer and Communication Security* (CCS), pp 281 – 289. October 2003.

[4] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, and Darko Stefanovic. Randomized Instruction Set Emulation. *ACM Transactions on Information and System Security*. In Press, 2005.

[5] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel (Second Edition)*. O'Reilly and Associates. 2002.

[6] Stephen W. Boyd and Angelos D. Keromytis. SQLrand: Preventing SQL Injection Attacks. *2nd Applied Cryptography and Network Security Conference* (ACNS). June 2004.

[7] Jedidiah R. Crandall, S. Felix Wu, and Frederic T. Chong. Experiences Using Minos as A Tool for Capturing and Analyzing Novel Worms for Unknown Vulnerabilities. *GI/IEEE SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment* (DIMVA). July 2005.

[8] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer-Verlag, 2002.

[9] Roman Danyliw. *CERT Advisory CA-2003-04 MS-SQL Server Worm*. January 2003. http://www.cert.org/advisories/CS-2003-04.html

[10] eEye Digital Security. *Sapphire Worm Code Disassembled*. January 2003. http://www.eeye.com/html/Research/Flash/sapphire.txt

[11] Intel Corporation. *Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference*. 1997. http://developer.intel.com/design/ pentium/manuals/24319101.pdf.

[12] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. *10th ACM International Conference on Computer and Communications Security* (CCS). October 2003.

[13] David Litchfield. *Variations in Exploit methods between Linux and Windows*. July 2003. http://www.ngssoftware.com/papers/exploitvariation.pdf

[14] The NASM Project. *The Netwide Assembler*. http://nasm.sourceforge.net/

[15] The Pax Team. *The Design and Implementation of PaX*. November 2003. http://pax.grsecurity.net/docs/pax.txt

[16] Julian Seward. *The Design and Implementation of Valgrind*. 2003. http://developer.kde.org/~sewardj/ docs-2.0.0/mc_techdocs.html

[17] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, Dan Boneh. On the Effectiveness of Address-Space Randomization. *11th ACM Conference on Computer and Communications Security*. October 2004.

[18] Solar Designer. *Return-to-libc Attack*. Bugtraq Mailing List. August 1997.

A. MicroVM Code

The MicroVM code is shown below using NASM assembly code [14]. For clarity, we use symbolic constants in this code; the appropriate values would be hard coded into the injected code by the worm during deployment. NUM_BYTES is the size of the worm execution buffer (22), DATA_OFFSET is the offset from the beginning of the worm code to the beginning of the data (a four-byte value), and REG_BYTES is the number of bytes used to store the worm registers (24).

```
_start:
    push ebp ; save frame pointer
    ; get location of stored worm registers
    mov ebp, WORM_ADDRESS + REG_OFFSET
    pop dword [ebp + DATA_OFFSET], ebp
    xor eax, eax ; eax is the IP into worm
    ; WormIP = eax (zeroing eax starts at the beginning)

read_more_worm:
    ; copy next NUM_BYTES into worm execution buffer
    cld
    xor ecx, ecx
    mov byte cl, NUM_BYTES
    mov dword esi, WORM_ADDRESS
```

```
    ; get WormIP (points at next instruction to fetch)
    add dword esi, eax
    mov edi, begin_worm_exec
    rep movsb
    ; change next WormIP to point to next block
    add eax, NUM_BYTES
    pushad ; save MicroVM registers

    ; load worm registers
    mov edi, dword [ebp + EDI_OFFSET]
    … ; do the same for esi, eax, ebx, ecx, and edx

begin_worm_exec:
    nop  ; Reserve NUM_BYTES using nops to leave
    nop ; room for worm code fragment
    … ; end of worm code space

    ; save worm registers
    mov [ebp + EDI_OFFSET],edi
    … ; do the same for esi, eax, ebx, ecx, and edx

    popad ; load MicroVM registers
    jmp read_more_worm
```

Acronym List

AES	Advanced Encryption Standard
API	Application Programmer Interface
ASR	Address Space Randomization
BAA	Broad Agency Announcement
BIND	Berkeley Internet Net Daemon
CERT	Computer Emergency Response Team
CSD	Calling Sequence Diversity
DOD	Department Of Defense
GAO	General Accountability Office
GDT	Genesis Diversity Toolkit
GNU	GNU's Not Unix
GSM	Groupe Special Mobile
HTTP	HyperText Transfer Protocal
ISR	Instruction Set Randomization
MAC	Message Authentication Code
OS	Operating System
PC	Program Counter
PHP	PHP: Hypertext Preprocessor
QOS	Quality Of Service
RISE	Randomize Instruction Set Emulation
SDR	Software-Defined Radio
SER	Simple Execution Randomization
SISR	Strong Instruction Set Randomization
SPEC	Standard Performance Evaluation Corporation
SQL	Structured Query Language
SRS	Self-Regenerative System
SSR	Stack Space Randomization
VM	Virtual Machine
XSS	Cross-Site Scripting