Security through Diversity

Leveraging Virtual Machine Technology

Genesis extends the traditional software development toolchain using application-level virtual machine technology to enable the practical realization of dynamic diversity transforms. Diversity, when judiciously applied, is a practical and effective defense against both return-to-libc and code-injection attacks.

Daniel Williams, Wei Hu, Jack W. Davidson, Jason D. Hiser, John C. Knight, and Anh Nguyen-Tuong
University of Virginia

Today's computer system networks have greatly increased productivity as well as overall quality of life. Such systems are ubiquitous, and our reliance on them to both control vital infrastructures and serve as common appliances for carrying out everyday tasks has made protecting them an international priority.

One major threat to our networked infrastructure's resilience is the software monoculture, the cyber analog of "putting all your eggs in one basket." Stephanie Forrest and her colleagues at the University of New Mexico observed the dangers a software monoculture presents and advocated introducing diversity to prevent large-scale attacks by forcing adversaries to tailor attacks to individual machines.[1]

Here, we present an overview of an approach we developed that modifies and extends the traditional software toolchain to apply dynamic diversity techniques and thus break the monoculture. This extended toolchain, called Genesis, applies diversity transformations at various points during the translation of a source program to an executable binary. Unlike existing toolchains, which treat the binary program as the final output, Genesis treats it as a vehicle for realizing diverse software system executions using Strata, an effective application-level virtual machine (VM). This lets software companies deliver the same binary to all customers but ensures that every binary execution will be diverse and that mounting a single attack against each executing instance is infeasible.

Diversity Overview

Diversity techniques aim to produce variants that are either immune to an attack—that is, that don't contain the targeted vulnerability—or that prevent the attack from succeeding even though the vulnerability is present. Most deployed diversity techniques are the latter type.

We can broadly classify diversity techniques into two complementary approaches: *design diversity* and *data diversity*.

Design Diversity

Design diversity has a long history in engineering, especially as regards developing safety-critical systems. Engineers in all fields recognize that defects can occur in their designs and have dealt with them using multiple parallel approaches to safe operation. To avoid common design defects, engineers now use different designs within systems. For example, many complex electrical systems are protected from failure using simple mechanical interlocks—engineers view the electrical and mechanical systems as different designs.

Ada Augusta, Countess of Lovelace, is generally credited with first suggesting using design diversity for software. The most prevalent form of software design diversity is *N*-version programming.[2] In an *N*-version system, developers write redundant software components to the same specification in the conventional manner. The premise behind design diversity is that separating methodology and programming teams will result in different faults in the different versions—although software will still fail, the hope is that different versions will fail at different times and in different ways.

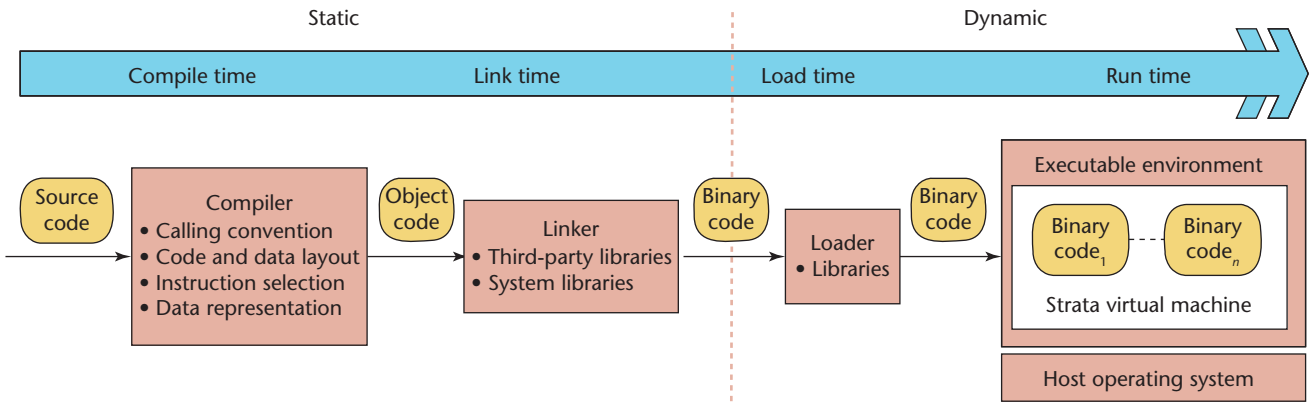

Figure 1. The Genesis toolchain, a traditional software toolchain extended with a virtual machine (VM). The Strata VM treats the binary as input and implements diversity transformations at runtime. Often, the diversity transform needs to be prepared at compile or link time for Strata to apply diversity at runtime.

Data Diversity

Data diversity is the concept that diversity in the data space (as opposed to the design space) can potentially avoid event sequences that lead to failure.[3] Applying data diversity changes the data that a program reads, causing the program to execute a different path and thereby possibly avoid a fault.

Data diversity's advantage over design diversity is that it lends itself to automation and is thus scalable. Examples include the notion of Heisenbugs that Gray reported for the Tandem Non-Stop operating system and techniques such as combining tree transformations, data, and code reordering for compilers.[1,4]

Data diversity doesn't remove vulnerabilities, it only makes them harder to exploit. Attacks to reverse the entropy diversity techniques provide—that is, de-randomization attacks—appear in the literature.[5,6] Genesis seeks to raise the bar for carrying out such attacks by applying a wide range of diversity transformations and increasing the resulting entropy level.

Genesis Software Development Toolchain

Figure 1 shows the Genesis software development toolchain. The multistep translation process from source code to an executing program occurs at four distinct epochs: compile time, link time, load time, and runtime. Each epoch lets developers apply transformations designed to introduce diversity. For example, a compiler might select from among multiple calling conventions, multiple ways of laying out code and data, or different instructions and registers. Similarly, the linker might select different implementations of a library to resolve calls to library functions (such as malloc/free, printf, threads, or signal handling).

A key design decision is choosing the appropriate epoch at which to apply a diversity transformation. A tension exists between applying a transformation early in the translation process versus late. Certain transformations must be applied early because the information required to implement them isn't available later in the process. For example, it's much more difficult to apply a transformation that affects the calling convention after a compiler produces machine code. On the other hand, applying a transformation that involves the calling convention at compile time, although simple to do, creates another problem: it produces multiple binaries, creating both manufacturing and distribution problems.

We believe deploying large numbers of different binaries would be infeasible, so we decided to forego applying a transformation at an epoch if doing so would produce a different binary. This design philosophy follows the one Windows Vista, Linux, and Mac OS X have all adopted in the form of address space randomization, which performs diversity transformation at load time (http://pax.grsecurity.net/docs/aslr.txt).

At first glance, the requirement to produce a single binary seems to severely restrict the possible diversity transformations. However, the advent of efficient application-level VM technology makes extending the traditional toolchain to include a runtime component practical.

Strata Application-Level Virtual Machine

The Strata VM, the runtime component we developed for the Genesis software development toolchain, implements software dynamic translation (SDT), which enables software malleability and adaptivity at the binary instruction level by providing facilities for runtime monitoring and code modification. SDT can affect an executing program by injecting new code, modifying existing code, or controlling

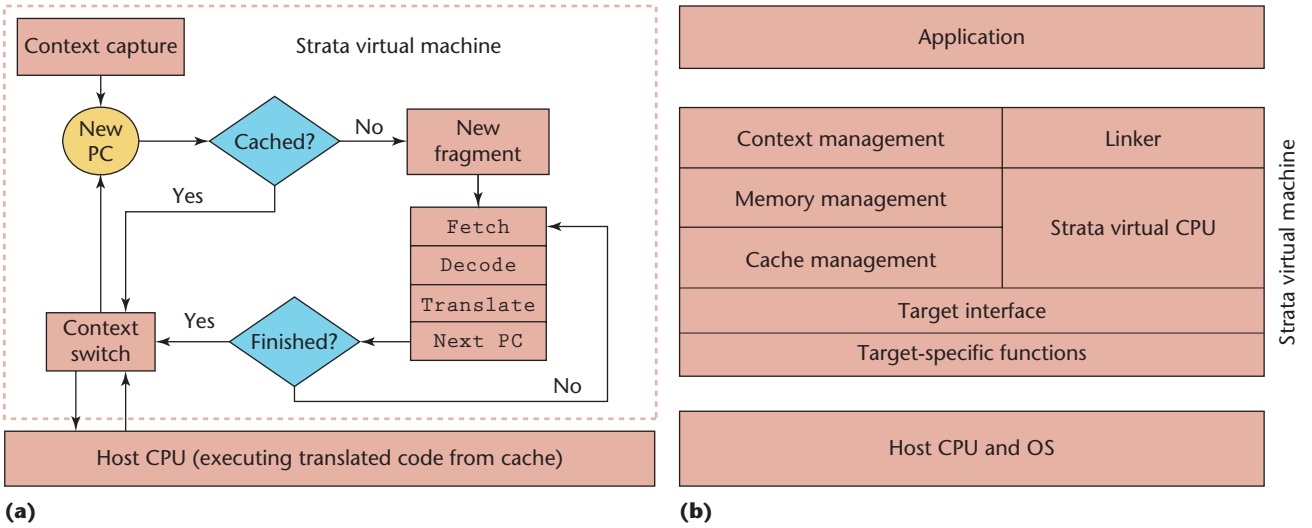

Figure 2. Strata architecture. (a) Strata operates by fetching, decoding, and translating an application's instructions into a software-managed cache. The virtual machine components `fetch`, `decode`, `translate`, and `next PC` are target-specific functions invoked through the target interface. (b) Strata sits between the application and the host CPU and comprises a set of target-independent common services (shaded boxes), a set of target-specific functions, and a target interface through which the two communicate.

program execution in some way. As Figure 2a shows, Strata is organized as a VM that mediates the execution of an application's instructions. Strata's design and implementation make it easy to reconfigure and retarget its services for new applications and computing platforms.

The Strata VM mediates application execution by examining and translating instructions before they execute on the host processor. Strata holds translated instructions in a Strata-managed *fragment cache*. The Strata VM, implemented as a co-routine resident with the application, takes control by capturing and saving the application context (program counter, condition codes, registers, and so on). Following context capture, the VM processes the next application instruction. If Strata has cached a translation for this instruction, a context switch restores the application context and begins executing cached translated instructions on the host machine.

SDT using Strata has helped successfully detect and prevent certain software exploits.[7,8] A simple example is using Strata to make the stack nonexecutable, defeating stack-smashing code-injection attacks. Without support from the hardware, SDT developers can prevent stack-smashing attacks completely by augmenting the Strata instruction-fetch mechanism. The developer merely overrides the fetch function with one that enforces the property that instruction fetches can't lie within the stack's address range.

Because it's flexible and modular, the Strata SDT platform can introduce various forms of diversity during execution. We chose the two techniques we describe in this article, *calling sequence diversity* (CSD) and *instruction set randomization* (ISR), to illustrate how we can apply powerful diversity transformations early (one at compile time, one at link time) yet deploy only a single binary. The transformations are also interesting because they address two common, complementary attack classes—return-to-libc and code injection.

Related Approaches

The most widely deployed diversity technique in commodity operating systems is *address space randomization* (ASR; http://pax.grsecurity.net/docs/aslr.txt). With ASR, the operating system diversifies a program's layout to make it difficult for attackers to predict the location of critical variables and functions. A major benefit of ASR is its high effectiveness-to-cost ratio. It thwarts a wide range of attack classes, including code-injection attacks and return-to-libc attacks, at a performance cost of essentially zero. ASR's main limitation is its susceptibility to brute-force attacks.[6]

Although we focus on diversity-based techniques for protecting binaries, other approaches have been widely deployed. A well-known example is StackGuard, a low-cost technique for protecting the stack's integrity that uses a "canary" to detect attempts to corrupt a return address or frame pointer.[9] Another more comprehensive defense is to enforce nonexecutable data pages, thus preventing the application from executing foreign code from the stack and other data regions. Hardware support for nonexecutable data pages is now commonly available for both Intel and AMD processors but is often omitted

from processors for low-cost, high-volume embedded applications.

The techniques we present both overlap and complement these approaches. As we refine and explore further techniques to leverage VM technology for security, we'll seek to characterize and study our techniques' benefits-versus-cost ratios, both in isolation and coupled with other approaches.

Calling Sequence Diversity

Without question, using standard conventions and compatible interfaces has great benefits. Indeed, software developers and users adhere to various conventions computer manufacturers and operating systems designers have established. The application binary interface (ABI) is a specification that describes key aspects of a machine's operation, such as its calling convention, how to make system calls, or OS-imposed data alignment restrictions. A binary that conforms to the ABI should produce the same results when run on a system that correctly implements the ABI. (An API similarly specifies conventions at a higher level—source code.)

Unfortunately, a single implementation of a standard convention gives attackers fertile ground for large-scale attacks. Return-to-libc attacks rely on such a situation—during these attacks, an attacker exploits a vulnerability to overwrite a return address, causing the application to call a library function (such as `system()` or `exec()`) with inappropriate or malicious arguments. A return-to-libc attack is powerful because it doesn't require the attacker to inject code, but rather uses code that's already present in the target program.

The return-to-libc exploit is possible because the attacker knows the calling sequence used to implement the calling convention. Large-scale attacks can thus occur because every instance of the vulnerable program uses the same calling sequence.

One way to thwart return-to-libc attacks that mount large-scale assaults on widely used vulnerable applications is to use a different calling sequence in each executing application. These sequences should be constructed to ensure that an attack that works against one binary won't work against another. To this end, we developed and implemented our automated CSD technique.

Our approach requires the call function to check a random "hidden" parameter that acts as a key and is unique for each application instance. Without the proper key, an attacker can't execute the function successfully. Although the attacker might compromise one executing instance of the application, the identical attack won't compromise other executions.

We can contrast CSD with an approach such as StackGuard.[9] The two approaches overlap because they both cover some return-to-libc attacks. StackGuard seeks to protect specific data structures, such as the return address or frame pointer, but can only prevent overwriting these structures, which doesn't prevent all call graph attacks. CSD soundly prevents any type of attack against the statically detectable call graph. Consequently, CSD can work with techniques such as StackGuard to provide stronger protection than either mechanism alone.

```
int main()
{
   foo(arg1, arg2);
}

        (1) int key;
        (2)
        (3) int main() {
        (4)   key = keygen(key, &main, &foo);
        (5)   foo(arg1, arg2, key);
        (6)   key = keygen(key, &foo, &main);
        (7) }

keygen(key, src, dst) {
        return key ⊕ key_src ⊕ key_dst;
}
```

Figure 3. Pseudocode for calling sequence. The Genesis compiler annotates the program with the location of key-generation operations. Later, Strata dynamically inserts key-generation and key-checking operations based on these annotations.

Implementation

To implement CSD, we must have cooperation between the compiler and the Strata VM. The compiler inserts code to maintain a 32-bit key value that indicates the current function that is executing. A program's initial key value is set in `main()`, which is computed randomly at load time. Before calling another function, the program computes the key parameter by XORing three parameters: the current key value, the key value associated with the calling function, and the key value associated with the called function. The goal is to make sure that the program follows the original static call graph—that is, that the program calls the called function from a valid source. Figure 3 shows the performed transformation.

When the program enters a new function (or is about to execute a system call instruction), Strata compares the expected key value with the key value just computed. A mismatch indicates that an attacker has tampered with the control flow. Note that key values aren't present in the application's memory space. Instead, Strata can precalculate $\texttt{key}_{\text{src}} \oplus \texttt{key}_{\text{dst}}$ and only enables an attacker to map the current function's key to the called function's key and back again. Consequently, keys are hard to extract from the program, and even if an attacker knows a key, he or she can't directly use it because the application can't modify it.

During normal operation, a program will have a key

value set to the key value of the destination function prior to the call. On line 4 in Figure 3, for example, the value is $\texttt{key}_{\text{main}} \oplus \texttt{key}_{\text{foo}}$, which is equal to the correct key value, $\texttt{key}_{\text{foo}}$. However, consider a case in which a vulnerability in function `foo` allows for a return-to-libc attack on a critical function such as `system()`. Because the key value would correspond to `foo`, it wouldn't match the key expected by `system()`.

Implementing the key-generation and verification technique we just described requires two complementary actions—one at compile time and one at runtime. The Genesis compiler brackets application function calls with code to generate the proper key values (lines 4 and 6 in Figure 3). Handling issues such as function inlining (which eliminates calls all together), tail call optimizations (which converts call instructions to jump instructions), and indirect function calls is easy because the compiler knows that a function call will occur regardless of the exact instruction sequence used to make it. Dealing with such constructs would be difficult without compiler assistance and highlights one advantage of our extended toolchain. The compiler also annotates functions to indicate that the application is using CSD. Note that CSD currently doesn't handle shared libraries or signals.

The second action occurs at runtime: Strata, when translating the code for a called function, extracts compiler-inserted annotations and dynamically generates code to compare the computed key with the expected key value. The key's location in the argument list is fixed (even for variadic functions) and conveyed to the dynamic system via annotation.

Circumventing CSD

As outlined thus far, a sophisticated attacker could try two obvious approaches to circumvent CSD. First, the attacker could try to guess or deduce the key necessary to enter a function. Because function addresses are associated with their corresponding key values randomly at runtime, this late binding makes guessing the correct values difficult. Moreover, each executing instance of the binary uses different values. In effect, each binary is using a different calling sequence, ensuring that a successful attack against one execution instance won't succeed against others.

Second, an attacker might attempt to bypass the checking code because the vulnerability lets the attacker jump to arbitrary addresses. We address this possibility by having Strata insert the checking code at runtime rather than having the compiler insert the code. Thus, even if the adversary attempts a return-to-libc attack that jumps to some address offset from the start of the function address, or even directly to a system call instruction, Strata, when translating the code, inserts the necessary checking code. Furthermore, because Strata manages access to the cached code, jumps into the middle of cached blocks aren't permitted.

In essence, CSD enforces a program's static control-flow graph to address the threat that return-to-libc attacks pose. It doesn't attempt to protect control-flow edges from dynamic constructs, such as function calls via compromised function pointers. Consequently, such attacks might thwart CSD protections.

A more common attack class is code injection. We'll look next at a diversity technique that addresses such attacks.

Instruction Set Randomization

Today, little diversity exists in instruction set architectures for general-purpose computing. Essentially, three instruction sets are in wide use—IA-32, SPARC, and PowerPC. Of these, IA-32 is used most widely. Again, this lack of diversity gives attackers an advantage—they can assume that their targets are executing IA-32 instructions—but has also motivated ISR.

Harold Thimbleby first proposed using randomization to create a unique instruction set as a technique for preventing the spread of viruses.[10] Later, researchers at the University of New Mexico[11] and Columbia University[12] independently proposed it as a method for protecting against code-injection attacks. Both groups implemented ISR prototypes for the x86 using emulation (Valgrind at New Mexico [http://valgrind.org] and Bochs at Columbia [http://bochs.sourceforge.net]).

Because both techniques used emulation, the overhead for translation from the synthetic instruction set to the actual machine's instruction set (decryption) was quite high. On CPU-bound benchmarks, the Columbia researchers reported runtime overhead at 25 times the native execution speed. On I/O-intensive programs such as FTP, the overhead was 1.34x. Based on their results, the Columbia researchers concluded that ISR would be feasible only with special hardware support.

One limitation of current ISR implementations is their incompatibility with programs that use self-modifying code or just-in-time compilation techniques (such as a Java VM). We don't address this problem but instead focus on leveraging our extended toolchain and employing a combination of static binary rewriting and software dynamic translation to improve on prior ISR implementations, algorithms, and efficiency.

Implementation

We describe our ISR implementation in detail elsewhere, including experimental evaluation using real-world exploits.[13] Instead of using XOR as our encryption/decryption function, we use the cryptographically strong 128-bit Advanced Encryption

Standard (AES) algorithm, which prevents known plaintext attacks. Our implementation also improves on the original algorithm by detecting attack code reliably instead of letting injected malicious code execute and crash the program.

Our ISR implementation requires cooperation between the linkage editor and Strata. In the link-edit stage, we annotate a program binary so that Strata can successfully encrypt and decrypt the program at runtime. We use Diablo, a link-time binary rewriting tool, to prepare a binary.[14] Taking all the object files as input, we

1a. mark any application and needed library code as "encryptable," which distinguishes between application code and the code Strata uses;
2a. append a global one-byte tag to each instruction; and
3a. pad the application as necessary to align basic blocks on 128-bit binary. This step is needed for our AES implementation.

At runtime, the Strata SDT system then

1b. loads and encrypts the application using AES;
2b. decrypts the application instructions one fragment at a time to prepare for execution;
3b. checks that the decrypted instructions are valid application instructions prior to execution by checking the tag; and
4b. removes any tags, deposits the validated instructions in the fragment cache, and then executes them. This amortizes the decryption cost over all future code-fragment executions.

Our ISR implementation requires two simple Strata extensions. First, we introduce an encryption feature that applies AES to the application text before Strata begins to execute an application. Second, we override Strata's default fetch mechanism. The new fetch method decrypts and verifies an instruction before calling the default target-machine fetch method.

Thwarting Code-Injection Attacks

Our implementation allows only encrypted instructions from the original application to execute. If an attacker attempts to inject a code sequence while the program is running, the Strata VM will detect the injected code during step 3b. The tag inserted in 2a won't match any attacker–inserted tag with a $1/256^n$ probability, where n is the number of sequential instructions analyzed. Note that using a uniform constant value for the tag is sufficient because the decryption step (2b) will yield a random tag. The tag allows Strata to detect the injected code; having this feature distinguishes our approach from other ISR techniques that don't detect code injections directly but instead rely on the high likelihood that a program will crash when executing randomized instructions.[11]

One limitation of our current ISR implementation is that it doesn't allow self-modifying code (SMC). Although most network-service applications don't use SMC, applications such as just-in-time compilers do. We're currently investigating ways that handle SMC safely but still provide the protection ISR affords. Finally, note that all ISR implementations (and data diversity in general) result in an application terminating. We're currently investigating recovery mechanisms to prevent such application-level denial-of-service attacks.

Protecting the VM

An obvious attack avenue with both techniques can occur via a vulnerability within the VM implementation. Strata has various mechanisms in place to ensure that it can't be compromised. For example, all of Strata's data structures are unavailable to the application because Strata turns off page read/write permissions. Strata also monitors the application for system calls that adjust page permissions and disallows any attempts to modify Strata-owned pages.

However, a flawed VM implementation can still introduce vulnerabilities. For instance, a flaw might cause Strata to not monitor a system call that might change page permissions in some unforeseen way. However, we believe such flaws are unlikely. Strata's implementation is only about 30,000 lines of code for three machine architectures, including security features. An active research area is to provide formal code verification to ensure that the code possesses critical security properties.

Evaluation

To illustrate the potential applicability of CSD and ISR, let's examine performance results and look at a summary of a red team evaluation.

Performance

We have working prototypes for both CSD and ISR. To illustrate the potential of VM-based techniques on real-world applications, we present performance data for ISR on two widely deployed server applications, the Apache Web server and the BIND domain-name server.

We measured our results using a 2.8-GHz P4 Xeon with 512 Mbytes of RAM. We measured the space overhead, the baseline SDT system's performance overhead (no ISR), and the SDT-based ISR system's overhead. We normalized each measurement to native execution performance (which uses no SDT system).

With respect to space overhead, the binary prepro-

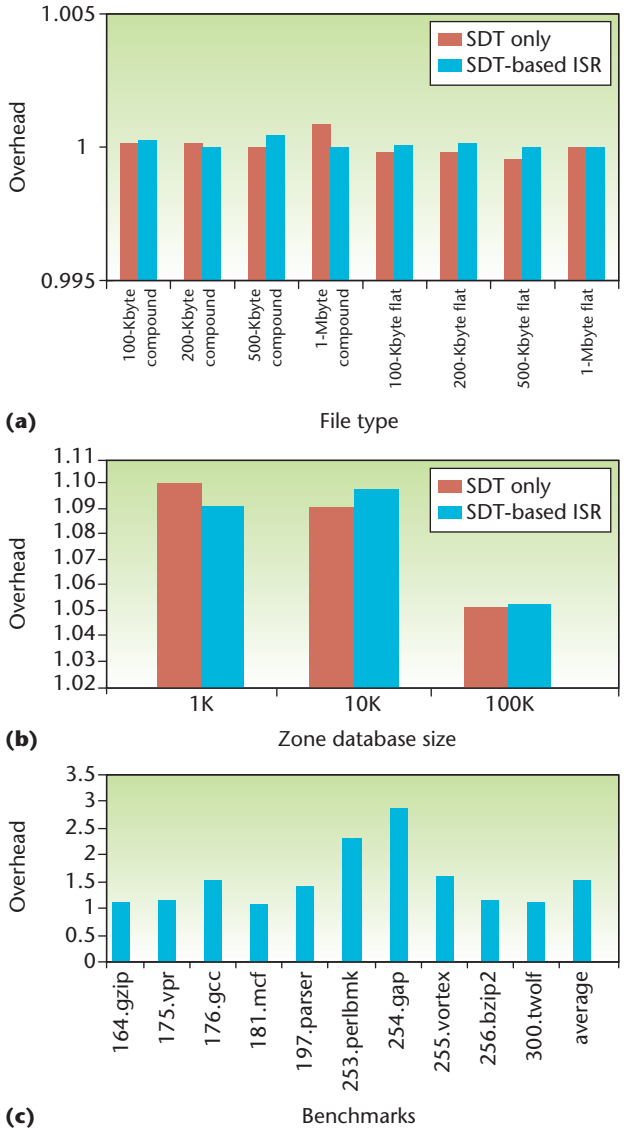

Figure 4. Genesis performance evaluation. We measured (a) Apache performance with instruction set randomization (ISR), (b) BIND performance with ISR, and (c) SPEC 2000 performance with calling sequence diversity.

cessing step increases both binaries by approximately 55 percent, primarily due to tag introduction.

To evaluate Apache's performance, we used the flood program to saturate the Web server and measured the number of client requests per second when Apache served a variety of Web pages. We found the performance overhead negligible compared to native execution (see Figure 4a).

To measure Bind's performance, 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 1,000 records, 10,000 records, and 100,000 records to represent differently sized organizations. Using *queryperf*, a DNS server performance-testing tool, we measured the number of queries processed per second that BIND ran under our SDT system with and without ISR enabled.

Figure 4b shows the measurement results. The overhead of querying the small- and mid-sized databases is roughly 10 percent over native execution, whereas the overhead for the larger database was 5 percent over native execution. In addition to measuring performance with these two server applications, we measured it using the SPEC CPU2000 benchmark suite.[14] In this case, ISR's overhead was only 17 percent over native execution.

Our CSD prototype is less mature and currently incurs average overhead of 54 percent over native execution on the SPEC 2000 benchmark suite. Figure 4c gives the performance breakdown. Performance analysis showed that the outlier programs (that is, *perlbmnk* and *gap*) have a high indirect branch frequency, causing excess overhead. The implementation naively checks the key value at every function entry point. We're investigating whether, without loss of security coverage, CSD could perform the checks only at system call boundaries.

Red Team Evaluation

DARPA, our project's sponsor, commissioned two independent security consulting firms (the red teams) to evaluate CSD and ISR and their implementation over the Strata platform. Our research group was the blue team. A neutral third party, the white team, mediated the exchanges between the blue and red teams. In the first exercise, we gave the red team a version of Apache seeded with vulnerabilities and their associated exploits, as well as documentation describing the algorithm and implementation behind ISR, CSD, and Strata. The red team had roughly one month to prepare for the actual one-day engagement, in which its primary goal was to break through 100 vulnerable Apache servers protected by CSD and ISR. The red team was unable to break into any of the variants.

For the second evaluation, we gave the red team all the information the first red team received, as well as full access to our source code and a comprehensive fault-tree system analysis. The goal for this exercise was to emulate a well-heeled adversary with access to insider information—that is, in-depth knowledge of the implementation and its potential weaknesses. The red team, in agreement with the white and blue teams, opted for an analytical study in which it systematically probed both the algorithms and their implementation. The red team was unable to find any major design or implementation flaws in our system.

Overall, the combined red team exercises yielded

a positive evaluation for our technology using both a black-box and a white-box approach, although the results are tempered by the limited time available to both red teams.

Based on our experience with the Genesis software development toolchain, we believe that synthetic dynamic diversity is a viable approach for providing security. We intend to investigate methods to broaden the attack classes our diversity techniques cover and develop algorithms for recovering and repairing programs using information from diversity protection mechanisms. □

Acknowledgments

This research was supported by the US National Science Foundation under grants CNS-0524432 and CNS-0716446, DARPA under grant FA8750-04-2-0246, and the US Department of Defense under grant FA9550-07-1-0532. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the sponsoring agencies.

References

1. S. Forrest, A. Somayaji, and D. Ackley, "Building Diverse Computer Systems," *Proc. 6th Workshop Hot Topics in Operating Systems* (HotOS-VI), IEEE CS Press, 1997, p. 67.
2. A. Avizienis and L. Chen, "On the Implementation of N-Version Programming for Software Fault Tolerance During Execution," *Proc. 1st IEEE Int'l Computer Science Applications Conf.*, IEEE Press, 1977, pp. 149–155.
3. P.E. Ammann and J.C. Knight, "Data Diversity: An Approach to Software Fault Tolerance," *IEEE Trans. Computer,* vol. 37, no. 4, 1988, pp. 418–425.
4. S. Bhatkar, R. Sekar, and D.C. DuVarney, "Efficient Techniques for Comprehensive Protection from Memory Error Exploits," *Proc. 14th Conf. Usenix Security Symp.*, Usenix Assoc., 2005.
5. A.N. Sovarel, D. Evans, and N. Paul, "Where's the Feeb? The Effectiveness of Instruction Set Randomization," *Proc. 14th Conf. Usenix Security Symp.*, Usenix Assoc., 2005.
6. H. Shacham et al., "On the Effectiveness of Address-Space Randomization," *Proc. 11th ACM Conf. Computer and Comm. Security* (CCS 04), ACM Press, 2004, pp. 298–307.
7. K. Scott and J.W. Davidson, "Safe Virtual Execution using Software Dynamic Translation," *Proc. 18th Ann. Computer Security Applications Conf.*, IEEE Press, 2002, pp. 209–218.
8. V. Kiriansky, D. Bruening, and S.P. Amarasinghe, "Secure Execution via Program Shepherding," *Proc. 11th Usenix Security Symp.*, Usenix Assoc., 2002, pp. 191–206.
9. C. Cowan and P. Wagle, "StackGuard: Simple Stack Smash Protection for GCC," *Proc. GCC Developers Summit*, 2003; www.gccsummit.org/2003.
10. H. Thimbleby, "Can Viruses Ever Be Useful?" *Computers and Security*, vol. 10, no. 2, 1991, pp. 111–114.
11. E.G. Barrantes et al., "Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks," *Proc. 10th ACM Conf. Computer and Comm. Security* (CCS 03), ACM Press, 2003, pp. 281–289.
12. G.S. Kc, A.D. Keromytis, and V. Prevelakis, "Countering Code-Injection Attacks with Instruction-Set Randomization," *Proc. 10th ACM Conf. Computer and Comm. Security* (CCS 03), ACM Press, 2003, pp. 272–280.
13. W. Hu et al., "Secure and Practical Defense against Code-Injection Attacks Using Software Dynamic Translation," *Proc. 2nd Int'l Conf. Virtual Execution Environments*, ACM Press, 2006, pp. 2–12.
14. B. De Bus et al., "Link-Time Optimization of ARM Binaries," *ACM SIGPLAN Notices*, vol. 39, no. 7, 2004, pp. 211–220.

Daniel Williams *is a PhD candidate in the Department of Computer Science at the University of Virginia. His research interests include compilers, virtual machines, and security. Contact him at dww4s@virginia.edu.*

Wei Hu *is a PhD candidate in the Department of Computer Science at the University of Virginia. His research interests include compilers, virtual machines, and security. Contact him at wh5a@virginia.edu.*

Jack W. Davidson *is a full professor in the Department of Computer Science at the University of Virginia. His research interests include embedded systems, security, and virtual machines. Davidson has a PhD in computer science from the University of Arizona. He is a member of the IEEE. Contact him at jwd@virginia.edu.*

Jason D. Hiser *is a research scientist in the Department of Computer Science at the University of Virginia. His research interests include compilers, security, and virtual machines. Hiser has a PhD in computer science from the University of Virginia. He is a member of the IEEE. Contact him at jdh8d@virginia.edu.*

John C. Knight *is a full professor in the Department of Computer Science at the University of Virginia. His research interests include all aspects of software dependability. Knight has a PhD in computer science from the University of Newcastle upon Tyne. He is a member of the IEEE. Contact him at knight@virginia.edu.*

Anh Nguyen-Tuong *is a senior scientist in the Department of Computer Science at the University of Virginia. His research interests include security and large-scale distributed systems. Nguyen-Tuong has a PhD in computer science from the University of Virginia. He is a member of the IEEE. Contact him at nguyen@virginia.edu.*