An Attack Simulator for Systematically Testing Program-based Security Mechanisms

Ben Breech
Computer and Info Sciences
University of Delaware
Newark, DE 19716
breech@cis.udel.edu

Mike Tegtmeyer
Army Research Lab
Aberdeen Proving Ground,
Maryland 21005-5068
mtegtmeyer@arl.army.mil

Lori Pollock
Computer and Info Sciences
University of Delaware
Newark, DE 19716
pollock@cis.udel.edu

Abstract

The use of insecure programming practices has led to a large number of vulnerable programs that can be exploited for malicious purposes. These vulnerabilities are often difficult to find during traditional software testing. In response to these difficulties, various program-based security mechanisms have been proposed to help protect potentially vulnerable programs. Testing these security mechanisms, however, also can be difficult and is currently rather adhoc.

In this paper, we describe the design, implementation, and evaluation of an attack simulator that enables the systematic and semi-automatic testing and evaluation of the effectiveness of current and future security mechanisms by automatically providing numerous contexts for testing the reliability of the mechanisms. Capable of automatically creating attacks on running programs by dynamically adding code (but not modifying existing code), the attack simulator can run in different modes and simulate attacks at various program points systematically. Through a case study, we demonstrate how our tool can be used to test two well-known security mechanisms for stack smashing attacks in several different testing modes.

1. Introduction and Motivation

The use of insecure programming practices has led to a large number of vulnerable programs that can be exploited in harmful ways with a variety of attacks [1, 2, 11, 20, 29]. Detecting these vulnerabilities during the software testing phase can be notoriously difficult as the required knowledge, time and other resources can be expensive. The goal of traditional correctness-based software testing is to expose faults in the component under test to increase confidence in the correctness of the code. Software testing is usually performed by generating a set of test cases, executing the component with each test case, and then verifying that the program behaves as expected. One challenge for security testing is that traditional software testing techniques tend to focus on common cases, i.e., those cases likely to be encountered by users. Security vulnerabilities, however, focus on the uncommon cases for which there are very few test cases. In response to these difficulties, various mechanisms have been proposed that attempt to detect and prevent an exploit from succeeding during program execution.

In this paper, we focus on *testing the security mechanisms* that have been implemented to protect against the exploitation of program vulnerabilities. Testing of these security mechanisms can be difficult and tedious, which leads to lowered levels of confidence in the reliability of the mechanism itself. The typical process is to first find a program with a known vulnerability and exploit, and then compile the program with the security mechanism in place. The vulnerability is then exploited to determine whether the mechanism can correctly detect the attack. Essentially, the exploit becomes one test case for testing the mechanism. This approach is not systematic, as one would normally like to test a wide variety of programs, as well as test numerous different potential attack points in each program. Gathering many programs with known exploits can be difficult because, as noted in [12], security organizations often refuse to release exploits, and the source code for vulnerable versions of a program are often removed to avoid accidental use.

Assuming the lack of testable programs could be overcome, the issue of performing systematic testing of the security mechanisms on the available vulnerable programs remains. The current testing process for security mechanisms focuses on the one or two functions that a known exploit abuses, ignoring other possible attack points in the program. If a flaw in the mechanism exists that causes protection to be unavailable at some potential attack points, that flaw can go unnoticed and lead to a false sense of confidence in the mechanism.

name	# funcs in actual prog	# funcs in benchmark	# funcs called	# funcs exploited
sendmail (s2)	357	5	5	2
sendmail (s4)	393	3	3	1
sendmail (s5)	672	3	3	1
bind (b3)	258	3	3	1
bind (b4)	258	6	6	1
wu-ftpd (f2)	196	6	3	2
wu-ftpd (f3)	153	6	6	1

Table 1. Motivating the need for systematic testing of mechanisms

As an example, consider Table 1 which shows a subset of benchmarks which have been used in buffer overflow studies [30]. The benchmarks are smaller versions of applications that have a documented history of vulnerabilities. The benchmarks have just enough of the original programs to showcase one vulnerability. In all cases, the number of functions that are affected by a given exploit is a small percentage of the total number of functions in the actual program. For example, only 2 functions contain an exploit to be used for testing a security mechanism in the `sendmail` benchmark, while there are 672 functions in the whole `sendmail` application. Manually inserting vulnerabilities in a large number of functions for testing more potential contexts for exploits would be tedious and error prone. Testing a security mechanism only against these known exploits would mean that a large percentage of the possible attack points and different contexts are not covered.

Covering only a small percentage of the possible attack contexts means faults can go undetected. Consider a simple mechanism that places a canary value around all character arrays, and then checks the canary values at the end of a function execution to ensure that no overflow occurred. This protection mechanism could easily be implemented in a source to source translator. If this mechanism is tested at only a few attack points (i.e., function call contexts), the effects of a compiler realigning local variables could easily go unnoticed, and the incorrect operation of the protection mechanism would go undetected, giving the tester false confidence in the reliability of the protection mechanism.

These considerations motivate the need for a testing tool that enables a more *systematic* approach to testing protection mechanisms for *program-based* attacks without having to obtain (or manually create) vulnerable programs. In this paper, program-based attack refers to attacks that are maliciously initiated on a program as input. Examples include stack smashing [1], attacks on function pointers [11], etc.

In this paper, we describe the design, implementation and evaluation of an attack simulator capable of automatically creating attacks on running programs by *dynamically adding code* (but not modifying existing code). The "program under test" in our system is the protection mechanism itself. A test case in our system is a tuple consisting of a compiled program P with the protection mechanism in place and input to the program P. Inputs used as test cases for program P are selected from the normal test suite for P to ensure proper coverage of the potential attack points for which the protection mechanism is being tested. Thus, any test case that would normally be used for testing the program P for correctness can be used; the simulator will simply insert attacks at different points and examine how the security mechanism reacts. For example, in the case of stack smashing attacks, protection mechanisms such as ProPolice [14] and RAD [10] could be tested by simulating the effects of a stack smashing attack on each function call executed in the program P and selecting a test suite for P that covers all function callsites in P.

Our previous paper [4] described the overall design of the framework for systematic testing in which the attack simulator works. This paper makes several contributions beyond our previous work, namely describing the details of the attack simulator design, implementation, and evaluation within the context of the framework. In addition, we present a case study demonstrating the testing of two well-known security mechanisms for stack smashing.

The key insight of our approach is the use of a dynamic compiler for simulating attacks to systematically *program-based protection mechanisms*. Our approach enables more systematic testing as different test scenarios, (e.g., different calling contexts, functions with different numbers/types of parameters) can be covered. This raises tester's confidence that the the security mechanism is reliable and operates correctly in a wide variety of contexts. We currently view our technique as being used primarily by developers of new security mechanisms. In the future, our approach could be integrated into the general software development process.

The rest of this paper is organized as follows. Section 2 describes the design of the attack simulator. Section 3 demonstrates how the attack simulator can be used to test actual security mechanisms, the evaluation of which appears in section 4. Section 5 describes related work and section 6 wraps up with conclusions and future work.

2. Automated Attack Simulation

2.1. Overall Design

The key insight behind the design of our attack simulator is that attacks against a running program can be simulated by *adding machine instructions* into the program itself. At no point are the instructions of the program itself modified. The inserted code does not depend on system or library resources. Our technique ensures that no functionality of the original program is removed or modified. This includes any

security mechanism added by the compiler. This capability is provided by dynamic compilers, such as DynamoRIO [5], Strata [24], and DynInst [7].

Generally, a dynamic compiler takes, as input, a compiled program and allows modifications or analysis of that program during runtime. The ability to modify code as the program executes enables an analysis or transformation module to take advantage of runtime information and react to changing environments. Most dynamic compilers allow the user to write their own modules to perform the analysis and modification. Our attack simulator is designed as one of these modules.

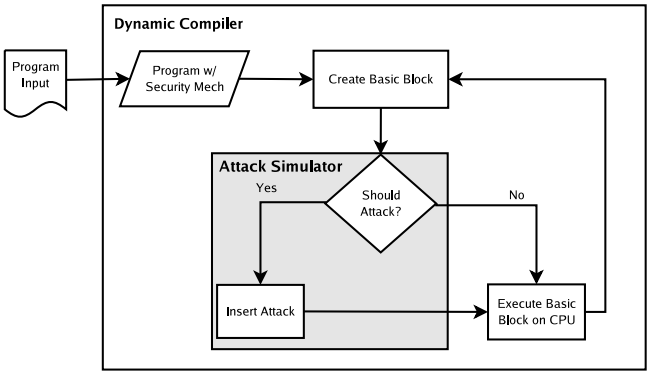

Figure 1. Attack Simulator Overview

Figure 1 shows a general overview of how a dynamic compiler and our attack simulator work together. The dynamic compiler uses a program compiled with the security mechanism as input (the program itself will typically have its own input as well). The dynamic compiler then enters a loop where it constructs sequences of machine instructions that would execute contiguously (i.e., basic blocks). The basic blocks are then given to our attack simulator which decides if an attack should be inserted into the block. If the simulator decides not to create the attack, the block is given back to the dynamic compiler without modification. If an attack should be generated, the simulator adds the appropriate instructions to the block and gives the block back to the dynamic compiler. The block is then given to the CPU for native execution. Once the CPU has executed the block, control goes back to the dynamic compiler, and the process repeats. More details about the construction of dynamic compilers can be found in [6, 17].

By basing the attack simulator on a dynamic compiler which can manipulate any program at runtime, we can enable systematic testing of program-based security mechanisms on a broad range of applications without restricting the testing to a small set of applications that contain a few known exploits. Attacks can be generated on a program without having to craft exploits by hand. Testers benefit from increased testing coverage without the tedious manual manipulation of programs. Consumers benefit from increased reliability of the security mechanisms through more thorough testing.

We have established several requirements for the automated attack simulator (and the framework that encompasses the simulator) [4]. These requirements include:

1. *Generality* – The simulator should be able to support a variety of source languages, different kinds of vulnerabilities and must be able to test a wide variety of program-based security mechanisms.

2. *Systematic testing* – The simulator must be able to insert attacks at any appropriate point (i.e., providing different contexts) in the program. Furthermore, the simulator must be able to handle different modes of attacks. This allows for more thorough and systematic testing of program-based security mechanisms.

3. *Automatic* – A tester only needs to specify the kinds of attacks that should be attempted and have the option of specifying the program points to be attacked. After that, the simulator should perform all of its tasks automatically without the need for user intervention.

4. *Robustness* – The simulator should accurately report the effectiveness of the security mechanism without generating false positives (e.g., reporting that the mechanism under test successfully protected the program where it actually did not).

5. *Low overhead* – The simulator must have reasonable overhead, in terms of both space and time, to be considered practical.

Some of these requirements are easily satisfied by using dynamic compilers. Since the input to a dynamic compiler is a compiled binary, the source language is largely irrelevant (except for some minor details such as the layout of function parameters), which helps satisfy the *generality* requirement. *Automation* is achieved in a straightforward manner through the implementation of the attack simulator itself. *Overhead* is incurred through using a dynamic compiler as well as the simulator itself. However, most dynamic compilers have techniques to decrease their overall overhead. This places most of the burden of reasonable overhead on the attack simulator itself. Overhead, identifying all appropriate attack points for testing in different contexts (*systematic testing*), and establishing the effectiveness of various mechanisms (*robustness*) will be discussed and evaluated in Section 4.

2.2. Testing with the Attack Simulator

During the testing process, a program is selected to be attacked by our simulator. The program need not contain an

actual vulnerability. The program, along with the protection mechanism, is then executed under the dynamic compiler with the attack simulator loaded. The test cases used with the program are selected from the normal test suite to ensure proper coverage of the attack points to be used in testing the mechanism under test (e.g., that all call sites where a protection mechanism for stack smashing should be tested are covered. During execution, the attack simulator examines the basic blocks it is given and inserts attacks appropriately. The decision to insert an attack is determined by the characteristics of the mechanism being tested. If the mechanism being tested detects the attack, the mechanism performs its designated actions, which will usually result in program termination. The fact that the mechanism successfully detected the attack is noted as part of the testing process. If the mechanism fails to detect the attack, the program will, by default, be terminated and the tester is notified of the failure.

2.3. Enabling Recovery for Test Efficiency

The default behavior of many security mechanisms is to terminate the program early since, if an attack occurred, the program is likely to be in an unknown state. In some instances, however, the attack simulator knows exactly what damage was done and can recover from the attack by undoing that damage. Thus, during our testing process, it is sometimes possible to recover from attacks and to continue testing on the same attacked program execution. Obviously, recovery cannot occur if the attack inserted by the simulator modifies any program data.

Recovery during testing is desirable as it enables more thorough testing of the mechanism with less executions of the program, thus saving on testing time. The simulator can continue inserting attacks, recovering from each of them, and allowing the program to continue executing and testing the security mechanism at additional attack points.

However, in some cases, it may be necessary to modify slightly the mechanism being tested to enable recovery during testing. For example, if the mechanism detects an attack and calls one of its routines that terminates the program, it would be necessary to modify that routine so the program does not abruptly exit. In this case, it still would be noted that the mechanism detected the attack during testing. Any such modification must be made carefully to avoid unintended consequences, such as introducing possible false negatives or false positives. The extent of the modification should be to avoid abrupt program termination or modification of program state.

3. Case Study: Testing Stack-Smashing Protection Mechanisms

In this section, we demonstrate the concepts presented in Section 2 with a particular attack, stack smashing, and how some mechanisms used to detect and prevent the attack can be tested. We chose stack smashing as it is a well understood, but still prevalent form of attack [9].

3.1. Overview of Stack Smashing

Stack smashing is one form of a buffer overflow attack, where an attacker gives more data to a program than its internal buffers can store. The result is that the program inadvertantly writes past the end of the buffer. In stack smashing, the buffers targeted by the attack are declared local to a function. Space for locally allocated function variables is allocated by the compiler on the call stack as part of the *activation record* (AR) of the function. Also included in the AR are the parameters passed to the function and the *return address*, i.e., where program control should go when the function is finished. By overflowing the buffer, it is possible to trick the program into executing arbitrary code. There are many tutorials [1, 13] that provide canned implementations of this kind of attack.

As an example, consider Figure 2 which shows the beginning of a function code along with the associated activation record. By convention, call stacks "grow down" (i.e., the next AR pushed is at a lower address than the previous AR). When function `foobar` is called, the caller pushes the parameters from right to left, onto the stack,[1] followed by the return address. The next item pushed is the saved frame pointer (fp). The frame pointer is used to determine offsets to function parameters and local variables. The saved fp is the caller's frame pointer. The callee will create its own fp as part of the function initialization. Finally, the function allocates space for the locally defined variables, in this case a 5 character array `buf1`, an integer `z`, and a 20 character array `buf2`.

Accessing the locations `buf2 [0]` through `buf2 [19]` would be legitimate as there is space allocated on the stack. However, accessing `buf [20]` is not legitimate and causes a bounds error. C does not check array bounds, however, so the access is permitted. The actual value used will be 20 bytes (20 `* sizeof (char)`) away from `buf2 [0]`. This address references the first byte of `z`. A value written into `buf2 [20]` will overwrite the first byte of `z` causing mysterious errors. This is an example of a buffer overflow.

By continuing to overwrite, we can write into any of the variables that are above `buf2`. In particular, we can get

[1] This is the standard C calling convention. Other languages, such as Pascal and Fortran push parameters left to right.

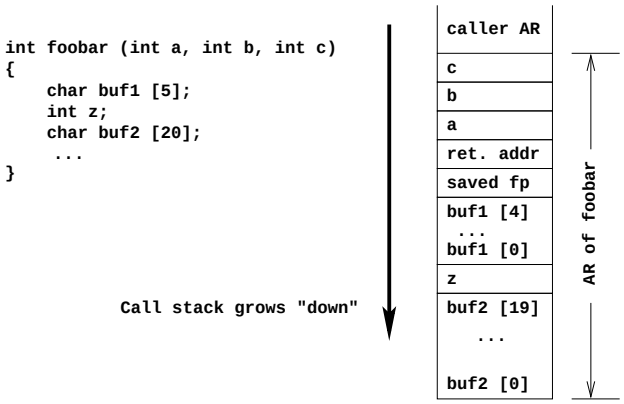

Figure 2. Example activation record

the the second byte of `z` by writing to `buf2 [21]`; `buf2 [24]` gives us `buf1 [0]`, and so on. Clearly we can place arbitrary values into `z` and also `buf1`. However, there is no reason to stop with overwriting `buf1`. Data can still be written beyond `buf1` and into the saved fp and, finally, the return address. By modifying the return address, an attacker can cause program control to jump to an arbitrary point and execute instructions when `foobar` returns. Typically, the input used to overflow the buffer includes code to spawn a shell process ("shell code"). The return address is modified to point to the shell code, which is stored on the stack and gets executed when the function returns.

3.2. Security Mechanisms Under Test

The basic idea of stack smashing is to overflow a buffer in a function to gain access to the return address. Everything between the start of the buffer and the return address is overwritten. Mechanisms, such as ProPolice and RAD, have been proposed to identify this anomalous behavior.

ProPolice [14] is a modification of the StackGuard [12] mechanism. Both mechanisms insert a canary value into the activation record of a function such that the canary value will be overwritten by any stack smashing attack. The canary can then be checked before a function return to verify that the canary was not modified. It is assumed that if the canary is unchanged, then the return address is also unchanged and is safe to use. ProPolice goes a step further by rearranging the activation record format for a function so that all buffers are above other variables. This prevents an attack where a buffer could be overflowed to modify a local pointer, but not reach the canary value. The modified pointer then could be used to modify the return address [8].

RAD [10] stores copies of the return address of a function in a special stack. When a function returns, its return address is compared against the address on top of the special stack. If they differ, then an attack has occurred.

3.3. Possible Testing Contexts

To use our attack simulator to test the ProPolice and RAD mechanisms, the simulator needs to insert code that simulates the behavior of a stack smashing attack. Stack smashing modifies the return address stored in the AR. A successful attack is executed when the function returns. As such, each function execution becomes a potential point of attack. Each function execution provides a different possible context for testing the protection mechanism because the AR can change from one call to the next. When a function is called, the attack simulator can insert instructions into the function code to simulate an attack at that call.

By default, the attack simulator will insert attacks into all of a program's call sites executed during the test input runs. This would include functions that may not necessarily have any potential security vulnerability. If static analysis [21, 25, 27] determines that some functions are not vulnerable, then they need not be protected, and the attack simulator can use this static information to avoid inserting attacks at those program points during execution.

3.4. Modes of Attack Simulation

Since modifying the return address is the ultimate goal, there are three different ways we can simulate an attack. Each has different tradeoffs and requires slightly different implementation. We designed and implemented three different modes of attack called *direct*, *tailored* and *random overflow*.

1. **Direct attack.** In this mode, the generated attacks directly modify the return address of the function without modifying any other data in the activation record. This is easily accomplished by inserting instructions immediately after the call instruction in the callee to modify its return address on the stack. We can set the return address to any location that we choose, which can help with recovery (see below).

   The result of the direct attack is identical to the final result of stack smashing (and many other attacks) since the return address is modified. The direct attack does not exactly simulate an attack, but is still useful for testing.

2. **Tailored attack.** Some protection mechanisms rely upon the general characteristics of buffer overflows rather than the final result to detect attacks, e.g., ProPolice places a canary value in the AR and checks this values for any modification. To test such mechanisms, the attack simulator has to tailor the attacks to the protection mechanism's mode of operation.

   In this mode, the attack simulator again modifies the return address in the callee code just after the call is

made. However, in addition to modifying the return address, the tailored attack also modifies any other data needed to trigger the protection mechanism. No other data in the AR is modified. This attack is tailored to a particular mechanism since the simulator requires exact knowledge of the mechanism. In our case study, we only need a tailored attack simulator for the ProPolice mechanism, since RAD only checks the return addresses. For testing ProPolice, the tailored attack overwrites this canary with a random value.

3. **Random overflow.** In this mode, the simulator attempts to create an actual buffer overflow by writing random numbers into a given buffer. The selected buffers are those used in potentially vulnerable library calls, such as `gets` or `strcpy`, that are susceptible to buffer overflows. The random overflow approach most closely simulates a real buffer overflow.

3.5. Recovering from Attacks

In the direct and tailored modes of operation, the attack simulator does not damage the data of the program. Only the return address, and other canary values are modified. As such, it is possible to recover from the attack to continue testing the protection mechanism with the same attacked program execution in both of these modes. However, recovery is not possible from an random overflow attack since it modifies program data, leaving the program in an unknown state.

For direct and tailored modes, recovery is straightforward. When attacking function f, the return address is first saved by the attack simulator on a stack it maintains. The return address of f is then replaced with the address of a special function which is not executed normally by the program. If the security mechanism under test successfully determines that an attack has taken place, the mechanism will perform its actions. If the attack is not detected, then when f returns, the special function is executed. In both cases, the result is recorded and instructions can be added during the execution to restore the correct return address from the attack simulator's stack and the program is allowed to continue as if no attack had occurred at that point. This process may require a slight modification to the protection mechanism to avoid it terminating the program.

Recovery allows more efficient testing of the protection mechanism since the program can continue normal execution after each attack. All the functions protected by the mechanism can be tested with only one execution of the program and one test case that covers all of the attack points (if available). Systematic testing without the recovery procedure would require executing the program once for each function protected by the mechanism, which will require more testing time.

3.6. Expected Testing Results

We would expect the ProPolice mechanism to discover all randomized overflow attacks and the tailored attacks. We would not expect ProPolice to detect the direct attacks since the canary value would be unchanged. RAD should be able to detect the direct attacks and the randomized overflow attacks in all attack contexts.

3.7. Simulating Other Attacks

In this paper, we focus on stack smashing to demonstrate the utility of the attack simulator. Other attacks are possible and work is currently progressing in implementing them. For example, attacks on function pointers [11] can also be simulated. Indirect calls are typically performed by loading the function address into a register and then issuing a call instruction using that register. The attack simulator can scan the code to find an indirect function call and note the register being used. The simulator can then find the load instruction that places the value into the register. The load instruction gives the location of the function pointer. A new instruction can be added just before the load to place a new value into the function pointer. This effectively simulates an attack that replaces the contents of the function pointer. Since function pointers are used in various contexts throughout a program, our attack simulator enables the protection mechanisms for vulnerabilities related to function pointer usage to be tested in these various contexts without testers having to manually manipulate the code to create these attacks.

4. Experimental Study

We performed an evaluation of the attack simulator by using it to generate stack smashing attacks for ProPolice and RAD. We have focused our evaluation of the attack simulator on several questions with respect to our requirements as discussed in Section 2.1.

1. *Systematic Testing - How effective is the attack simulator at providing systematic testing of a given security mechanism?*
2. *Robustness - How reliable is the attack simulator in reporting accurate test results for the different security mechanisms being tested?*
3. *Overhead - Is it practical to use the automatic attack simulator during the testing cycle?*

With regard to Question 1, the effectiveness of the attack simulator in providing systematic testing is determined by how well it can identify potential attack points (i.e., contexts) and provide the tester with different options for testing. For the direct and tailored modes, the attack simulator

should be able to identify and attack all functions that were protected by the mechanism. For the randomized buffer overflow mode, the attack simulator should be able to identify and overflow the appropriate buffers.

For Question 2, the robustness of systematic testing based on the attack simulator is determined by using the attack simulator to test the security mechanisms and comparing how well the security mechanism identifies the automatically generated attacks.

The testing cost of using the attack simulator (Question 3) can be answered by taking measures of the time and space requirements of the simulator in different modes while actually systematically testing a given mechanism. Ideally, the attack simulator in each mode would add very little overhead.

4.1. Setup

For our current implementation, we use DynamoRIO [5] as the dynamic compiler. Use of DynamoRIO follows very closely to Figure 1. The user writes a client module that DynamoRIO then loads and calls when particular events occur. The attack simulator, which is written as a client module of DynamoRIO, is called when a basic block is created, which allows for attacks to be inserted.

We implemented the attack simulator as three separate DynamoRIO modules, one for each operating mode — direct, tailored, and random overflow. For the direct mode, the attack simulator inserts instructions immediately after the call to an attacked function (and before the AR is set up) to modify the return address. The ProPolice tailored mode is implemented similarly with the addition of modifying the canary value. The randomized buffer overflow simulator is set up to attack functions that call `strcpy`.

One difficulty we experienced with the implementations was determining calls to `strcpy`. The code for `strcpy` is stored in the `libc` system library, so the address for `strcpy` cannot be determined until the program is executing. Further complicating the problem, DynamoRIO will stitch together blocks so a call instruction in the shared library may not be seen by the attack simulator. To get around these issues, we used a wrapper function for `strcpy` that called `strcpy` and then executed three assembly instructions that have no net effect, but which the attack simulator could identify by monitoring executed instructions to determine that the appropriate call has occurred. This approach violates our automation requirement since it involves user intervention to insert the wrapper call. Future work on the attack simulators will provide a better solution.

We also ran experiments using the recovery mechanism, discussed in Section 3.5. The recovery mechanism was applied to the direct and tailored attack modes for ProPolice. No recovery was performed for RAD as it was implemented within DynamoRIO. Performing both the mechanism and recovery within DynamoRIO could have compromised the validity of the mechanism.

We executed several medium-sized applications to determine the effectiveness of the attack simulator in testing the security mechanisms. The applications and their characteristics are listed in Table 2. The simple benchmarks used in the introduction were too small to be of use in our experimental study. We chose two example mechanisms to test – ProPolice and RAD. We used the implementation of ProPolice found in GCC 4.1. We implemented RAD as a DynamoRIO module. There is no interference between the RAD module and the attack simulator. The applications were compiled with the different security mechanisms in place and then executed under DynamoRIO with each of the modules for the different modes of the attack simulator loaded. The programs were executed using typical test cases that would normally be used for testing the program. The test cases do not attempt to exploit any vulnerability. This helps demonstrate the utility of our attack simulator since it enables testing of security mechanisms without first having to find an exploitable program.

Program	Description	Source LOC	funcs
008.espresso	Boolean function minimizer	9,844	363
026.compress	Compression program	1,043	19
130.li	Lisp interpreter	4,888	366
space	ESA ADL Interpreter	6,230	136

Table 2. Subject Programs for Testing Security Mechanisms

4.2. Data and Analysis

Table 3 shows the data we collected to use in our analysis of the systematic testing capability of the attack simulators[2] (Question 1). The *number-of-dynamic-function-calls* column contains the total number of calls to application functions during execution of the test case. The *number-of-calls-attacked* column gives the number of function calls that occurred during execution and were attacked by the attack simulator. This number was computed by running the application with the attack simulator, but disabling the actual insertion of attacks. The attack simulator recorded the number of times it would have inserted an attack. This allowed the data to be collected with one run of the program. The last two columns provide a sense of the function coverage of the test case used. The number of static callsites is determined by examining the source code looking for application calls. The number of callsites covered is the number

[2]We call each mode of the attack simulator an attack simulator in the remainder of this paper for brevity.

of static callsites that were executed by the program with that test case. The difference between the two columns reveals how many callsites were not covered by the test case.

name	# dynamic func. calls	# calls attacked	# static callsites	# callsites covered
space	180	179	367	51
026.compress	250911	250910	30	8
008.espresso	1439761	1439760	1279	412
130.li	1.6×10^9	1.6×10^9	944	535

Table 3. Systematic testing with Attack Sim.

Table 3 shows the effectiveness of the attack simulators in enabling systematic testing. The numbers shown are those for ProPolice and are identical for the direct and tailored attack modes. The simulators were effective in finding attack points, having found all but one for each application. In each case, the only function called, but not attacked, was `main`. The exact reason for this is unclear but seems to be an artifact of how DynamoRIO handles the startup of `main`, rather than a flaw in the attack simulator. Since RAD was implemented as DynamoRIO module, our implementation of RAD also missed protecting `main`. After accounting for that, the numbers for RAD would be identical for those in the table. The last 2 columns give function coverage information for the test case. The coverage is adequate for a single test case, although a tester would certainly want to execute more test cases to cover other callsites. It is clear that the attack simulator is effective in identifying potential points of attacks.

Table 4 shows the test results for the two security mechanisms against each attack simulator mode (Question 2). The name column gives the name of the application used. The ProPolice mechanism was tested with the direct and tailored attack simulator modes. These are abbreviated d and T in the table. RAD was tested with the direct mode. The *number-of-calls-attacked* column presents the number of *unique* function calls attacked. In the current implementation the attack simulator inserts an attack into a function when that function is first executed. Without recovery the first such attack would terminate the program. Hence the numbers reported here are different than those reported in Table 3. The *attacks-detected* columns contain the number of attacks the mechanism being tested discovered. The numbers in this table were computed by instructing the attack simulator to attack one function, running the program, and then repeating this process for each function executed by the test case.

The mechanisms performed as expected, with ProPolice being unable to recognize the direct attacks, but performing well for the tailored attacks. Similarly, RAD detected the direct attacks. All possible attacks were accounted for so it is clear that the simulator accurately reported the test results (Question 2).

Name	# calls attacked	# attacks detected		
		ProPolice		RAD
		d	T	
space	35	0	35	35
026.compress	5	0	5	5
008.espresso	170	0	170	170
130.li	117	0	117	117

Table 4. Testing Results

Table 4 shows the effectiveness of the different protection mechanisms. The RAD mechanism was very effective in detecting the attacks. ProPolice was 100% effective when the attack modified the canary value. However, it offers no protection against the direct attack mode. This would represent the case where an attacker correctly guessed the canary value or found some way to bypass the canary value entirely. The chances of that happening in practice may be low. It would be up to the users of the mechanism to decide if that is acceptable.

We ran the randomized buffer overflow attack simulator on the space program, which called `strcpy` in 8 of the executed functions. In all cases the destination buffer was filled with random values and overflowed past its correct size in an attempt to reach the return address. Both ProPolice and RAD detected 4 of the attacks. We investigated why the other 4 attacks were not detected and discovered that destination buffers in those attacks were actually global variables. Global variables are not allocated on the stack and cannot be used for a stack smashing attack, so there is no reason to expect the mechanisms would catch these attacks. Both mechanisms were effective in detecting the attacks they should have.

Table 5 shows the costs of using the attack simulators during the testing process. The spatial requirements for the attack simulator is insignificant as space is only required for the code module, so we focus only on the time needed to test the mechanisms. The timing results are reported in seconds. The second column gives the execution time of each program executed normally, i.e., without DynamoRIO running the program. The other columns give information on the costs of performing the actual tests of the particular security mechanism. To perform systematic testing, each program was run multiple times, once per executed function with the attack simulator inserting attacks against that function. The times for each execution were recorded from which an average, minimum and maximum time could be calculated. The last column gives the time needed to test the mechanism when recovery is used. This is the time of a *single* program execution and includes attacking *all function calls*, as listed in Table 3.

In all cases (except `space`), the average time to test 1 function call was less than the time needed to actually execute the program. `space` is an extremely short lived pro-

name	exec. time w/out attacks	avg time to test 1 call	min/max time to test 1 call	# calls tested	total time test all calls	time for all calls (w/ recovery)
ProPolice						
space (d)	0.004	0.072	0.043/0.090	35	2.54	0.13
space (T)	0.004	0.085	0.052/0.108	35	2.98	0.14
008.espresso (d)	0.621	0.44	0.022/0.93	170	75.6	1.4
008.espresso (T)	0.621	0.45	0.047/1.06	170	76.5	1.7
026.compress (d)	0.16	0.14	0.025/0.46	5	0.72	0.5
026.compress (T)	0.16	0.16	0.054/0.46	5	0.80	0.6
130.li (d)	52.1	3.67	0.028/194	117	430	–
130.li (T)	52.1	3.68	0.045/192	117	430	–
RAD						
space	0.004	0.071	0.042/0.088	35	2.48	–
008.espresso	0.621	0.46	0.021/4.4	170	77.7	–
026.compress	0.16	0.15	0.024/0.50	5	0.75	–
130.li (RAD)	52.1	3.66	0.027/194	117	428	–

Table 5. Testing Costs

gram so the overhead from starting DynamoRIO becomes significant. The maximum time occurred when the function being attacked was executed towards the end of the program execution. In all cases, the typical testing time is not prohibitively long.

The recovery mechanism performed well, enabling systematic testing of the mechanisms at all executed function calls with one execution of the program. `130.li` acted abnormally during testing with recovery enabled. We believe the reason for this is that `130.li` makes use of the `setjmp/longjmp` functions, which enables programs to immediately jump from the currently executing function back to an earlier function on the call stack. Future versions of the attack simulator will handle this behavior correctly.

4.3. Evaluation Summary

Our attack simulator was clearly effective in providing systematic testing (Question 1). The simulator correctly identified the possible attack points (i.e., different possible contexts) related to our case study for stack smashing. The results obtained by our attack simulator for testing two different security mechanisms were as expected (Question 2). Clearly, our attack simulator can be used to provide systematic and accurate testing of other program security mechanisms. These results enable testers to feel confident that the implementations of the security mechanisms that we tested are indeed correctly handling all the different contexts of function callsites covered by the test cases input to the programs used in the testing of the mechanisms. Finally, the cost of using the attack simulator for testing is reasonable (Question 3). When applicable, the recovery mechanism provides for more efficient testing as only one execution of the program can cover many different contexts for testing.

5. Related Work

As far as we are aware, this is the first effort in providing a systematic method of testing security mechanisms. There has, however, been work done in testing the effectiveness of various mechanisms [28]. In that work, a wide variety of attacks, including stack smashing, heap overflows, and overflows to modify pointers, were applied against different mechanisms, including ProPolice. They found that mechanisms tended to perform poorly, which, perhaps, is to be expected since many of the mechanisms were not designed to counter some of the attacks. For instance, ProPolice provides no protection against heap overflows.

Other work has focused upon preventing exploits in a general fashion. Array bounds checking [3, 19, 23] will prevent buffer overflow attacks from occurring, but can be expensive to use in production code. Other dynamic techniques focus on sandboxing a program so that out of place system calls can be avoided [16], examining the pattern of system calls to detect anomalies [15, 26], or providing run time type checking [22]. Static techniques include modifying the C language itself to avoid most security problems [18, 22], using programmer annotations to find potential vulnerabilities [21], or checking for vulnerabilities by checking string manipulations [27].

6. Conclusions and Future Work

We have presented the design, implementation and evaluation of an attack simulator capable of automatically inserting attacks into executing programs by adding to (but not modifying) the program instructions. The simulator can be used to systematically test program-based security mechanisms on a broad range of applications without restricting the testing to a small set of applications that contain a few known exploits. Attacks can be generated on a progrma

without having to craft exploits by hand. Testers benefit from increased testing coverage without the tedious manual manipulation of programs. consumers benefit from increased reliability of the security mechanism through more thorough testing. We showed how the attack simulator is effective in identifying and attacking different program contexts for stack smashing attacks, and systematic testing two well-known security mechanisms for stack smashing attacks.

Our future work will concentrate on designing a better method for recovery, and improved identification of program points for the randomized buffer overflow attack. We are working on simulating other kinds of attacks, including attacks against function pointers in order to broaden the kinds of security mechanisms that can be tested by our system.

References

[1] AlephOne. Smashing the stack for fun and profit. `http://www.insecure.org/stf/smashstack.txt`.

[2] anonymous. Once upon a free ()... `http://www.phrack.org/show.php?p=57\&a=9`.

[3] R. Bodík, R. Gupta, and V. Sarkar. ABCD: Eliminating array bounds checks on demand. *Programming Language Design and Implementation*, 2000.

[4] B. Breech and L. Pollock. A framework for testing security mechanisms for program-based attacks. *Software Engineering for Secure Systems*, 2005.

[5] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization*, 2003.

[6] D. L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, M.I.T., 2004.

[7] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *Journal of Supercomputing Applications and High Performance Computing*, 2000.

[8] Bulba and Kil3r. Bypassing StackGuard and StackShield. `http://www.phrack.org/phrack/56/p56-0x05`.

[9] C. C. Center. `http://www.cert.org`.

[10] T. Chiueh and F. Hsu. RAD: A compile-time solution to buffer overflow attacks. *International Conference on Distributed Computing Systems*, 2001.

[11] M. Conover. w00w00 on heap overflows. `http://www.w00w00.org/files/articles/heaptut.txt`.

[12] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. *USENIX Security Symposium*, 1998.

[13] DilDog. The tao of windows buffer overflow. `http://www.cultdeadcow.com/c{D}c_files/c{D}c-351`.

[14] H. Etoh and K. Yoda. GCC extension for protecting applications from stack-smashing attacks. `http://www.research.ibm.com/trl/projects/security/ssp/`, 2000.

[15] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. *Symposium on Security and Privacy*, 2004.

[16] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. *Network and Distributed Systems Security Symposium*, 2004.

[17] D. Grove and M. Hind. The design and implementation of the Jikes RVM optimizing compiler. OOPSLA'02 Tutorial, 2002.

[18] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. *USENIX Annual Technical Conference*, 2002.

[19] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. *Automatic and Algorithm Debugging*, 1997.

[20] Klog. Frame pointer overwrite. `http://www.phrack.org/show.php?p=55&a=8`.

[21] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. *USENIX Security Symposium*, 2001.

[22] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. *Principles of Programming Languages*, 2002.

[23] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. *Network and Distributed Systems Security Symposium*, 2004.

[24] K. Scott and J. Davidson. Strata: A software dynamic translation infrastructure. *Workshop on Binary Translation*, 2001.

[25] J. Viega, J. Bloch, Y. Kohno, and G. McGraw. `ITS4:` a static vulnerability scanner for C and C++ code. *ACSAC*, 2000.

[26] D. Wagner and D. Dean. Intrusion detection via static analysis. *Symposium on Security and Privacy*, 2001.

[27] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. *Network and Distributed Systems Security Symposium*, 2000.

[28] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. *Network and Distributed Systems Security Symposium*, 2003.

[29] R. Wojtczuk. Defeating Solar Designer non-executable stack patch. `http://www.insecure.org/sploits/non-executable.stack.problems.html`.

[30] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *Foundations of Software Engineering*, 2004.